Chapter 28.7 (N9). Webcam Snaps

Over the next few chapters, I'll look at using non-standard input devices, including the gamepad, and the P5 data glove (chapters 11 and 12), but I'll start with the webcam.

The webcam is, perhaps suprisingly, a great 'building block' for creating unusual input devices. The trick is to use it to deliver images (e.g. of the user's hand or face) to the application, which extracts information from them to act as the input.

I'll explain the image processing aspects of this approach in chapter 10 when I code an 'arm waving' device. It allows me to navigate around a 3D scene using nothing but my hand (no keyboard or mouse is required). This chapter concentrates on how to generate the webcam snaps used in the image processing stage.

I'll examine and compare two ways of capturing images from a webcam: with TWAIN (the Technology Without An Interesting Name), and JMF (Java Media Framework). The aim is to grab images as quickly as possible, then display them in rapid succession in a JPanel. The panel output includes the number of pictures displayed so far and the average time to take a snap, information that'll help me judge the two technologies.

Figure 1 shows the TWAIN and JMF applications.

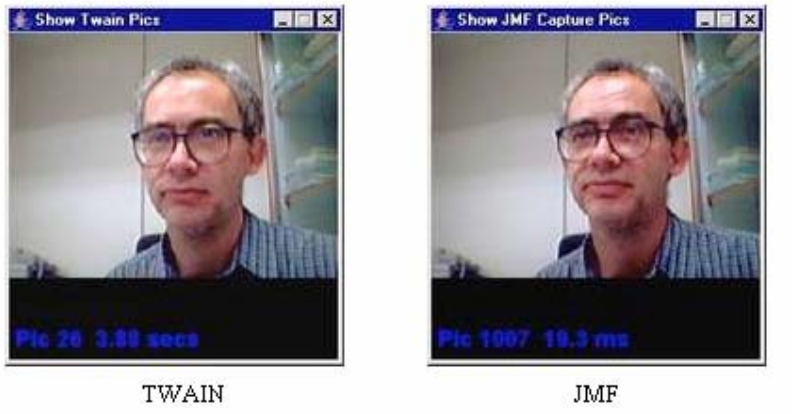

Figure 1. TWAIN and JMF Webcam Movies.

The TWAIN text in Figure 1 is "Pic 26. 3.88 secs", and the JMF message is "Pic 1007. 19.3 ms". This highlights the crucial difference between the two approaches – TWAIN takes a lot longer to generate a picture; it's 150-200 times slower.

1-2. Webcam Snaps with TWAIN

The TWAIN specification defines an API for obtaining images from scanners and digital cameras (http://www.twain.org/). There are implementations for Windows and the Mac, but UNIX/Linux isn't supported due to TWAIN's requirement that the capture device offers a dialog box that exposes its features. SANE (Scanner Access

Now Easy) is a popular image acquisition API for UNIX/Linux, described at http://www.sane-project.org/.

I'm using Morena 6.3.2.2, a commercial Java interface for TWAIN (it also supports SANE); a 30-day evaluation copy is available from http://www.gnome.sk/Twain/jtp.html. Another good commercial product, JTwain (http://asprise.com/product/jtwain/), offers a similar 30-day evaluation version. A free alternative, also called JTwain, was developed by Jeff Friesen in a series of articles at java.net (http://today.java.net/pub/a/today/2004/11/18/twain.html).

The Windows installation of Morena involves three JAR files (morena.jar, morena_windows.jar, and morena_license.jar). They should be added to Java's classpath, or copied into <JAVA_HOME>\jre\lib\ext and <JRE_HOME>\lib\ext.

TWAIN offers a simple abstraction for image acquisition, based on a *source* (the capture device) and a *source manager* which handles the interactions between the application and the source. As a consequence, the two most important Morena classes are TwainSource and TwainManager.

The main stages in a TWAIN-enabled application are quite simple:

1. Use TwainManager to select a source (a TwainSource object).
2. Get/set the source's capabilities.
3. Acquire images from the source.
4. Close down the source and manager, and exit.

3. Displaying Pictures using TWAIN

The ShowPics application uses TWAIN to grab pictures from the webcam, and displays them in a JPanel. The class diagrams for the program, with only the public methods visible, are shown in Figure 2.

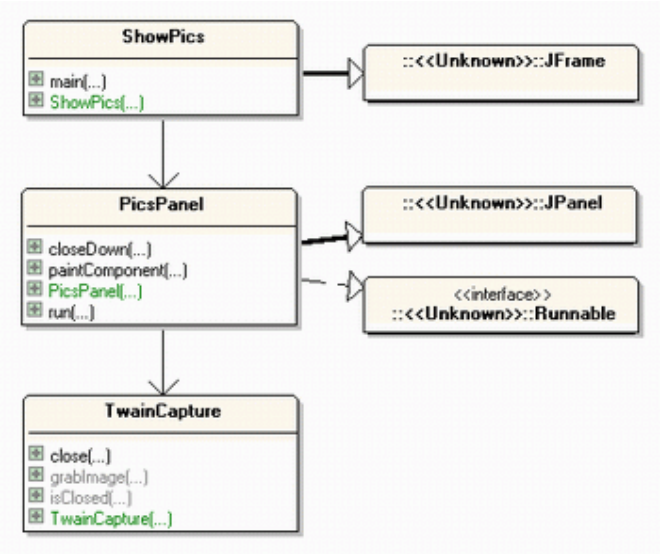

Figure 2. Class Diagrams for ShowPics.

PicsPanel is threaded so it can keep repeatedly calling grabImage() in the TwainCapture object. A grabbed image is drawn into the JPanel.

The only thing of note in ShowPics (the top-level JFrame) is that clicking its close box triggers a call to closeDown() in PicsPanel. This in turn calls close() in TwainCapture to close the link with the webcam.

3.1. Snapping a Picture Again and Again and ...

PicsPanel's run() method is a loop dedicated to calling grabImage() and to calculating the average time to take a snap.

```
// globals
private static final int DELAY = 1000;  // ms

private BufferedImage image = null;
private TwainCapture camera;
private boolean running;

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


public void run()
/* take a picture every DELAY ms */
{
  camera = new TwainCapture(SIZE);
     // must link to Twain device in same thread that uses it
```



```
  long duration;
  BufferedImage im = null;
  running = true;

  while (running) {
    long startTime = System.currentTimeMillis();
    im = camera.grabImage();   // take a snap
    duration = System.currentTimeMillis() - startTime;

    if (im == null)
      System.out.println("Problem loading image " + (imageCount+1));
    else {
      image = im;   // only update image if im contains something
      imageCount++;
      totalTime += duration;
      repaint();
    }

    if (duration < DELAY) {
      try {
        Thread.sleep(DELAY-duration); // wait until DELAY has passed
      }
      catch (Exception ex) {}
    }
  }

  // saveSnap(image, SAVE_FNM);   // save last image
  camera.close();   // close down the camera
}  // end of run()
```

Each iteration of the loop is meant to take DELAY milliseconds. The time to take a snap is stored in duration, 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.

The DELAY value used in run() is 1000 ms (1 second), which I chose by examining the statistics output to the screen when the program is executing. A delay of 1 second makes the webcam 'movie' run at 1 frame/second (1 FPS) or slower, which is inadequate for movie emulation.

Morena requires that the TWAIN source 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 usually crashes, and often causes Windows to start acting strangely. Consequently, all interactions with the TWAIN device are localized inside run().

Terminating the Application

The single-threaded requirement affects the way that the application is closed. When the user presses the close box in the JFrame, closeDown() is called in PicsPanel:

```
public void closeDown()
{
  running = false;
  while (!camera.isClosed()) {
    try {
      Thread.sleep(DELAY);  // wait a while
    }
```



```
    catch (Exception ex) {}
  }
}
```

closeDown() sets running to false, then waits for the camera to close. When running is false, the loop in run() will eventually finish, and TwainCapture.closed() will be called just before run() exits. closed() sets a boolean to true inside TwainCapture, which allows TwainCapture.isClosed() to return true. Only then will closedDown() return, permitting the application to terminate.

This approach means that the program's GUI will 'freeze' for 1-2 seconds when the close box is clicked, since closeDown() blocks the GUI thread while it waits.

The more usual way of coding termination behaviour is to put a call to TwainCapture.close() inside closeDown(). Although this enables the application to exit quickly, it also frequently crashes the JRE, and makes Windows unresponsive. This is a consequence of manipulating the TWAIN source in the GUI thread rather than in the PicsPanel thread.

Painting the Panel

The paintComponent() method draws the webcam picture in the panel, and writes some statistics near the bottom.

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  g.drawImage(image, 0, 0, this);   // draw the snap

  // write statistics
  g.setColor(Color.blue);
  g.setFont(msgFont);
  if (imageCount > 0) {
    double avgGrabTime = (double) totalTime / (imageCount*1000);
    g.drawString("Pic " + imageCount + "  " +
                 df.format(avgGrabTime) + " secs",
                 5, SIZE-10);  // bottom left
  }
  else  // no image yet
    g.drawString("Loading...", 5, SIZE-10);
} // end of paintComponent()
```

The panel has fixed dimensions (SIZE*SIZE), which explains the use of SIZE in the drawString() calls. There's no need to scale the image in paintComponent(), since grabImage() in TwainCapture does it.

paintComponent() is called when the application is first made visible, which occurs before any images have been retrieved from the source device. In that case, paintComponent() draws the string "Loading..." at the bottom left of the panel.

Saving a Snap

It's very simple to save a snap to a file by using the ImageIO.write() method:


```
private void saveSnap(BufferedImage im, String fnm)
// Save image as JPG
{
  System.out.println("Saving image");
  try {
    ImageIO.write(im, "jpg", new File(fnm));
  }
  catch(IOException e)
  {  System.out.println("Could not save image");  }
}
```

saveSnap() is called with the current BufferedImage, and a filename.

3.2. The TWAIN Capture Device

The TwainCapture constructor carries out the first two steps in using a TWAIN device:

1. Use TwainManager to select a source (a TwainSource object).
2. Get/set the source's capabilities.

```
// globals
private int size;       // x/y- dimensions of final BufferedImage
private double scaleFactor;  // snap --> returned image scaling

private TwainSource source;
private MediaTracker imageTracker;
private boolean closedDevice;


public TwainCapture(int sz)
{
  size = sz;
  scaleFactor = 0;   // dummy initial value

  source = null;
  imageTracker = new MediaTracker( new JPanel() );
  closedDevice = true;   // since device is not available yet

  source = chooseSource();
  // source = defaultSource();  // simpler alternative
  System.out.println("Using source " + source);

  // hide the TWAIN source user interface
  source.setVisible(false);
  closedDevice = false;
}  // end of TwainCapture()
```

The chooseSource() method selects a suitable source device and, TwainSource.setVisible(false) ensures that its features dialog box won't be displayed.

The MediaTracker object created in the constructor is used later to monitor the image acquisition in grabImage().

chooseSource() selects a TWAIN source based on whether it has a controllable GUI, which means that the setVisible() call in the constructor will have an effect.

```
private TwainSource chooseSource()
{
  TwainSource src = null;
  try {
    if (!TwainManager.isAvailable()) {
      System.out.println("Twain not available");
      close();
      System.exit(1);
    }

    TwainSource[] srcs = TwainManager.listSources();
    System.out.println("No. of Twain Sources: " + srcs.length);
    if (srcs.length == 0) {    // no sources
      close();
      System.exit(1);
    }

    int guiIdx = -1;   // index position of GUI controllable device
    for (int i=0; i < srcs.length; i++) {
       System.out.println("" + (i+1) + ". Testing " + srcs[i]);
       if (srcs[i].getUIControllable()) {
         guiIdx = i;
         System.out.println("  UI is controllable");
       }
    }
    if (guiIdx == -1) {  // no source is GUI controllable
      System.out.println("No controllable source GUIs");
      close();
      System.exit(1);
    }
    else
      src = srcs[guiIdx];
  }
  catch (TwainException e)
  { System.out.println(e);
    close();
    System.exit(1);
  }

  if (src == null) {
    close();
    System.exit(1);
  }

  return src;
}  // end of chooseSource()
```

The core actions of the method are to call TwainManager.listSources() and examine each of the TwainSource objects in the returned array.

chooseSource() is considerably lengthened by the need to handle errors. It's important to always close the source manager before exiting, or the OS may be affected. close() is:

```
synchronized public void close()
```



```
{ try {
    TwainManager.close();
  }
  catch (TwainException e)
  { System.out.println(e); }
  closedDevice = true;
}
```

close() and grabImage() are synchronized so that it's impossible to close the source while an image is being snapped. In fact, this behaviour is already impossible due to the coding style used in PicsPanel's run(), but I've left the synchronization in place as an extra safeguard.

An Alternative Way to Choose a Source

On a machine with a single capture device, which allows its features dialog box to be made invisible, the generality of chooseSource() is overkill. defaultSource() selects a source by retrieving the default one:

```
private TwainSource defaultSource()
// use the default TWAIN source
{
  TwainSource src = null;
  try {
    src = TwainManager.getDefaultSource();
  }
  catch (TwainException e)
  { System.out.println(e);
    close();
    System.exit(1);
  }
  return src;
}
```

This method can be called in the constructor instead of chooseSource():

```
source = defaultSource();
```

Grabbing an Image

Taking a snap is a matter of passing the TWAIN source object to Toolkit.createSource(), and waiting for image to be fully loaded.

```
synchronized public BufferedImage grabImage()
{
  if (closedDevice)
    return null;

  Image im = Toolkit.getDefaultToolkit().createImage(source);
  imageTracker.addImage(im, 0);
  try {
    imageTracker.waitForID(0);
  }
  catch (InterruptedException e) {
    return null;
```



```
  }
  if (imageTracker.isErrorID(0))
    return null;

  return makeBIM(im);
}
```

To make it easier for the snap to be manipulated, it's converted to a BufferedImage before being returned:

```
private BufferedImage makeBIM(Image im)
{
  BufferedImage copy = new BufferedImage(size, size,
                                  BufferedImage.TYPE_INT_RGB);

  // create a graphics context
  Graphics2D g2d = copy.createGraphics();

  // image --> resized BufferedImage
  setScale(g2d, im);
  g2d.drawImage(im,0,0,null);
  g2d.dispose();
  return copy;
}  // end of makeBIM()
```

There are numerous ways of formatting a BufferedImage: TYPE_INT_RGB assumes that there's no alpha component in the image. An alternative is to employ TYPE_3BYTE_BGR which supports dynamic texturing in OpenGL v.1.2 (and above) and D3D. TYPE_3BYTE_BGR would be a good idea if the image was going to be used as a texture in Java 3D.

setScale() resizes the image by modifying the graphics context 's scale factor:

```
private void setScale(Graphics2D g2d, Image im)
{
  if (scaleFactor == 0.0) {  // scale not yet set
    int width = im.getWidth(null);     // get the image's dimensions
    int height = im.getHeight(null);
    if (width > height)
      scaleFactor = ((double) size) / width;
     else
      scaleFactor = ((double) size) / height;
  }
  g2d.scale(scaleFactor, scaleFactor);  // scale the context
}
```

When setScale() is called the first time, it calculates a scale factor that will fit the image into the JPanel's dimensions (size*size). Later calls use that value to resize all the grabbed images.

4. TWAIN Timing Tests

Although the TWAIN approach works, its speed is somewhat less than spectacular. It takes several seconds for the device to be initialized (2-5 seconds). The time seems longer when TwainManager.listSources() is used instead of TwainManager.getDefaultSource(), but it's hard to judge since I only have two TWAIN devices attached to my machine.

The first call to TwainCapture.grabImage() takes several seconds (2-5 seconds) to retrieve the first image after the initialization has finished. This means there's a possible 10 second wait before the first image appears in the JPanel.

Subsequent calls to grabImage() gradually become faster, reaching a 'gallop' at a snap every 2-3 seconds.

A slow frame rate may not necessarily be a bad thing, since it depends on the requirements of the application. For instance, the need to reduce network bandwidth usage may make a reduced frame rate acceptable. Nevertheless, it would be better to use a technology where fast snap speeds were at least possible.

The rate set in PicsPanel's run() is one picture every 1000 ms (1 second), which is very optimistic. Since a snap really takes 2-3 seconds, the sleep code in run()'s loop never has an opportunity to be executed.

5. Webcam Snaps with JMF

I revisit the application in this section, replacing TWAIN with the Java Media Framework (JMF), more specifically the JMF Performance Pack for Windows v.2.1.1e (`http://java.sun.com/products/java-media/jmf/`). The overall class structure is much the same as before, as Figure 3 indicates.

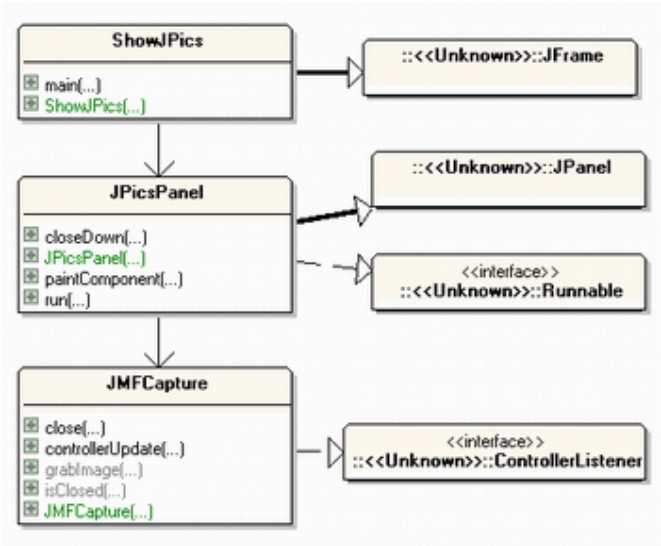

Figure 3. Class Diagrams for ShowJPics.

The class names have been changed (to protect the innocent), but ShowJPics is actually identical to ShowPics.

PicsJPanel is threaded in the same way as PicsPanel since it repeatedly calls grabImage() in the JMFCapture object.

JMFCapture contains the JMF code for accessing the capture device, but externally it looks just like TwainCapture, offering the same method prototypes.

5.1. Again Taking Pictures Again and Again

The run() method in JPicsPanel is similar to the one in PicsPanel:

```
// globals
private static final int DELAY = 40;  //ms

private JMFCapture camera;


public void run()
/* take a picture every DELAY ms */
{
  camera = new JMFCapture(SIZE);

  long duration;
  BufferedImage im = null;
  running = true;

  while (running) {
    long startTime = System.currentTimeMillis();
    im = camera.grabImage();  // take a snap
    duration = System.currentTimeMillis() - startTime;

    if (im == null)
      System.out.println("Problem loading image " + (imageCount+1));
    else {
      image = im;   // only update image if im contains something
      imageCount++;
      totalTime += duration;
      repaint();
    }

    if (duration < DELAY) {
      try {
        Thread.sleep(DELAY-duration);  //wait until DELAY time passed
      }
      catch (Exception ex) {}
    }
  }

  // saveSnap(image, SAVE_FNM);   // save last image
  camera.close();    // close down the camera
}  // end of run()
```

One reason for the similarity is that the JMF source is manipulated solely from the panel's thread, in the same way as the TWAIN source in PicsPanel. This makes JPicsPanel's closeDown() method identical to the one in PicsPanel:

```
public void closeDown()
{
```



```
  running = false;
  while (!camera.isClosed()) {
    try {
      Thread.sleep(DELAY);
    }
    catch (Exception ex) {}
  }
}
```

This has the same disadvantage as the coding in PicsPanel: when the user clicks on the close box, the application 'freezes' for 1-2 seconds while closeDown() waits for the run() loop to terminate, and JMFCapture.close() to finish. The more obvious approach of calling JMFCapture.close() directly from closeDown() will cause the OS to start behaving badly.

An important change is the value of DELAY – 40 ms, compared to 1 second in PicsPanel. Timing tests show that the JMF can capture an image every 30 ms on average. The remaining 10ms gives the run() loop time to sleep while the panel is being redrawn.

5.2. The Capture Device with JMF

The steps in taking a snap with JMF are more involved than in TWAIN:

1. Get a media locator for the specified capture device.
2. Create a player for the device, putting it into the *realized* state. A player in the realized state knows how to render its data, so can provide rendering components and controls when asked.
3. Create a frame grabber for the player.
4. Wait until the player is in the started state.
5. Initialize a BufferToImage object.
6. Start grabbing frames, and converting them to images.
7. Close the player to finish.

The code corresponding to stages 1-5 is commented in the JMFCapture constructor below:

```
// globals
private static final String CAP_DEVICE =
            "vfw:Microsoft WDM Image Capture (Win32):0";
            // common name in WinXP

private int size;       // x/y- dimensions of final BufferedImage

private Player p;
private FrameGrabbingControl fg;
private boolean closedDevice;


public JMFCapture(int sz)
{
```



```
  size = sz;
  closedDevice = true;   // since device is not available yet

  // link player to capture device
  try {
    MediaLocator ml = findMedia(CAP_DEVICE);  // stage 1

    p = Manager.createRealizedPlayer(ml);     // stage 2
    System.out.println("Created player");
  }
  catch (Exception e) {
    System.out.println("Failed to create player");
    System.exit(0);
  }

  p.addControllerListener(this);

  // create the frame grabber (stage 3)
  fg = (FrameGrabbingControl) p.getControl(
            "javax.media.control.FrameGrabbingControl");
  if (fg == null) {
    System.out.println("Frame grabber could not be created");
    System.exit(0);
  }

  // wait until the player has started (stage 4)
  System.out.println("Starting the player...");
  p.start();
  if (!waitForStart()) {
    System.err.println("Failed to start the player.");
    System.exit(0);
  }

  waitForBufferToImage();  // stage 5
}  // end of JMFCapture()
```

findMedia() calls CaptureDeviceManager.getDeviceList() to get information on the capture devices registered with JMF, then cycles through it looking for the named device.

```
// globals
private static final String CAP_LOCATOR = "vfw://0";


private MediaLocator findMedia(String requireDeviceName)
{
  Vector devices = CaptureDeviceManager.getDeviceList(null);

  if (devices == null) {
    System.out.println("Devices list is null");
    System.exit(0);
  }
  if (devices.size() == 0) {
    System.out.println("No devices found");
    System.exit(0);
  }

  for (int i = 0; i < devices.size(); i++) {
    CaptureDeviceInfo devInfo =
```



```
          (CaptureDeviceInfo) devices.elementAt(i);
      String devName = devInfo.getName();
      if (devName.equals(requireDeviceName)) {   // found device
        System.out.println("Found device: " + requireDeviceName);
        return devInfo.getLocator();   // this method may not work
      }
    }

    System.out.println("Device " + requireDeviceName + " not found");
    System.out.println("Using default media locator: " + CAP_LOCATOR);
    return new MediaLocator(CAP_LOCATOR);
  }  // end of findMedia()
```

If the desired device is found, then a MediaLocator object is created with CaptureDeviceInfo.getLocator(); a media locator is similar to a URL but for media devices. If the device isn't found, then the code uses the default locator string, "vfw://0".

There's a few issues with this approach, one being the choice of device name to pass to findMedia(). Windows XP and 2000 almost always name their capture device "vfw:Microsoft WDM Image Capture (Win32):0". Another common name is "vfw:Logitech USB Video Camera:0" for Logitech devices; Windows 98 employs "vfw:Microsoft WDM Image Capture:0".

Another problem is that a device must be registered with JMF before it can be manipulated by the API. Registration is carried out with JMF's Registry Editor, in the "Capture Devices" tab (see Figure 4).

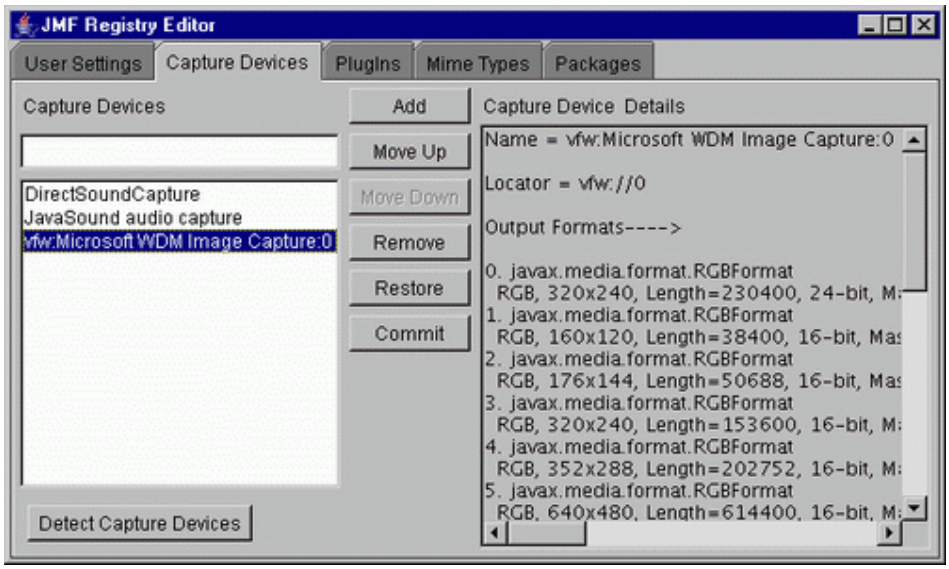

Figure 4. Registering Capture Devices in JMF.

Pressing the "Detect Capture Devices" button generates the device's name, media locator URL, and the supported image formats. This information is also available by clicking on the device name in the left hand list.

Unfortunately, there's no way to trigger this registration process from within a user program. However, if the webcam is present when JMF is installed, it will be registered automatically as part of the installation process.

Waiting for a BufferToImage

I was surprised to discover that the initialization phase may need several attempts to create a BufferToImage object.

waitForBufferToImage() calls hasBufferToImage() repeatedly until it returns true; usually 2-3 tries are necessary. hasBufferToImage() takes a snap, then checks if the resulting Buffer object really contains something.

```
// globals
private double scaleFactor;  // snap --> final image scaling
private BufferToImage bufferToImage = null;


private boolean hasBufferToImage()
{
  Buffer buf = fg.grabFrame();     // take a snap
  if (buf == null) {
    System.out.println("No grabbed frame");
    return false;
  }

  // there is a buffer, but check if it's empty or not
  VideoFormat vf = (VideoFormat) buf.getFormat();
  if (vf == null) {
    System.out.println("No video format");
    return false;
  }

  System.out.println("Video format: " + vf);
  int width = vf.getSize().width;     // the image's dimensions
  int height = vf.getSize().height;
  if (width > height)
    scaleFactor = ((double) size) / width;
  else
    scaleFactor = ((double) size) / height;

  // initialize bufferToImage with the video format info.
  bufferToImage = new BufferToImage(vf);
  return true;
}  // end of hasBufferToImage()
```

Aside from FrameGrabbingControl.grabFrame() returning null, it may return a Buffer object with no internal video information. Only when there's an actual VideoFormat object can a BufferToImage object be created. It's also then possible to extract the format's size in order to calculate a scale factor for later snaps.

Grabbing an Image

grabImage() captures a snap as a Buffer object by calling FrameGrabbingControl.grabFrame(), converts it to an image, and then to a BufferedImage.

```
synchronized public BufferedImage grabImage()
{
```



```
  if (closedDevice)
    return null;

  // grab the current frame as a buffer object
  Buffer buf = fg.grabFrame();
  if (buf == null) {
    System.out.println("No grabbed buffer");
    return null;
  }

  // convert buffer to image
  Image im = bufferToImage.createImage(buf);
  if (im == null) {
    System.out.println("No grabbed image");
    return null;
  }

  return makeBIM(im);  // image --> BufferedImage
}  // end of grabImage()
```

makeBIM() is similar to the same-named method in TwainCapture.

Closing the Capture Device

close() calls Player.close() to terminate the link to the capture device:

```
synchronized public void close()
{  p.close();
   closedDevice = true;
}
```

close() and grabImage() are synchronized so that it's impossible to close the player while a frame is being snapped. In fact, JPicsPanel already makes this impossible since close() is only called when run()'s snapping loop has finished. But I've left the synchronization in place just so I can feel a little safer; I did the same thing in TwainCapture.

6. Comparing TWAIN and JMF Capture

TWAIN isn't capable of generating sequences of captured images at high frames rates. It manages only one new frame every 3.5 seconds, compared to JMF's one image every 20 ms!

Both approaches are slow to start, taking several seconds to finish their initialization phase, but TWAIN also takes an age to obtain the first snap.

TWAIN is a Windows/Mac protocol, but JMF wins for portability since it also supports UNIX/Linux.

Morena 6, the Java TWAIN API I employed, is commercial software, while JMF is free. However, there is a freeware Java wrapper for TWAIN, described at http://today.java.net/pub/a/today/2004/11/18/twain.html.

TWAIN doesn't need a separate registration process in order to find capture devices at run time. JMF applications depend on the JMF registry knowing about devices, which requires the user to manually add new devices via the editor.

The TWAIN API is simpler to use than JMF because it focuses on image grabbing, while JMF's supports time-based multimedia. However, there are numerous JMF examples which explain these features, including the collection at Sun's JMF website (`http://java.sun.com/products/java-media/jmf/2.1.1/solutions/`). A good book on JMF, which also has several chapters on Java 3D is:

> *Java Media APIs: Cross-Platform Imaging, Media and Visualization*
> A. Terrazas, J. Ostuni, and M. Barlow
> Sams, 2002
> `http://www.samspublishing.com/title/0672320940`

One of its longer examples is a 3D chat room which combines Java 3D and JMF.

On balance, I'll be using JMF rather than TWAIN to deliver images to my "arm waving" application in the chapter 10. JMF is fast enough for my needs, portable, and free.

7. QTJ

Another way of implementing capture is to use *QuickTime for Java* (QTJ). QTJ provides an object-based Java layer over the QuickTime API, making it possible to play QuickTime movies, edit and create them, capture audio and video, and perform 2D and 3D animations. QuickTime is available for the Mac and Windows. Details about QTJ's installation, documentation, and examples can be found at `http://developer.apple.com/quicktime/qtjava/`.

Chris Adamson talks about video and audio capture in chapter 6 of *QuickTime for Java: A Developer's Notebook* (http://www.oreilly.com/catalog/quicktimejvaadn/). He's somewhat hampered by the fact that QTJ doesn't supports an on-screen component for its SequenceGrabber class; SequenceGrabber offers a grabPict() method, but there's no efficient way of getting that image onto the screen (Adamson mentions a 1 frame/second drawing rate).

A solution was found by Jochen Broz after the book's publication, and posted to the quicktime-java mailing list (http://lists.apple.com/archives/QuickTime-java/2005/Nov/msg00036.html). He bypasses the inefficiencies of grabPict() and generates a BufferedImage directly from the SequenceGrabber frame. Adamson explains the details in a note at http://www.oreillynet.com/mac/blog/2005/11/capturing_to_the_screen_with_q.html, and reports tests showing frame rates of 32 FPS, which is a frame captured every 30 ms or so; that's slower than JMF but still reasonable for many applications.

7. Other Uses for Webcam Snaps

Chapter 10's image processing is utilized in a 3D setting, but the code is virtually unchanged when employed in a 2D game. You can find details in the online chapter

28.85, "Waving a Magic Wand" at my *Killer Game Programming in Java* website (http://fivedots.coe.psu.ac.th/~ad/jg/). The user waves a 'magic wand' in front of a webcam, and an on-screen wand moves in a corresponding manner, and shoots magic bolts.

Another application of this chapter's frame grabbing technique is to paste the images, in rapid succession, onto a Java 3D 'move screen'. This is the subject of two articles I wrote for O'Reilly's website, one using JMF (http://www.onjava.com/pub/a/onjava/2005/06/01/kgpjava.html), the other QTJ (http://www.onjava.com/pub/a/onjava/2005/06/01/kgpjava_part2.html). They can also be found as chapters 28.3 and 28.5 at my *Killer Game Programming in Java* site.

A third application domain for webcams are as the eyes of robots. In the article "Futurama: Using Java Technology to Build Robots that can See, Hear Speak, and Move", Steve Meloan describes the robotics work of Simon Ritter, a Technology Evangelist at Sun (http://java.sun.com/developer/technicalArticles/Programming/robotics/). The robots utilize Lego Mindstorms running leJOS, an open source 14 KB JRE available from http://lejos.sourceforge.net/. The mix of Java technologies includes speech recognition, synthesis, and image processing. For instance, a PC takes webcam snaps of a ball, the images are analyzed, and the resulting data guides the robot to the ball.

Ritter initially used a TWAIN interface much like the approach in this chapter, but found it slow (he averaged 1.5 frames/second). He then moved to JMF, but using a slightly different technique than I've outlined. The capture device is accessed through a CaptureDeviceManager and MediaLocator object, but a PushBufferStream is connected to the device to make it act as a data source. A call to PushBufferStream.read() returns immediately with a Buffer object containing a captured image.

The drawback with this *push* technology is that read() returns the first picture from a stream of images generated by the camera, a stream which may have started to be filled as soon as the camera was switched on. That means that read() may retrieve an image that's several seconds out of date. My JMF code utilizes *pull* technology, which snaps an image only when requested, so ensuring that it's current.

The leJOS site is a good place to start finding out about Java Lego robotics. LeJOS was updated at the end of 2006 to support Lego NXT, the exciting next generation of Lego Mindstorms, launched in September 2006.

