Chapter NUI-2. Webcam Snaps

The webcam is a great 'building block' for creating novel forms of user input. At the lowest level, the camera repeatedly delivers images (e.g. of the user's hand or face) to a processing stage, which employs a range of image processing and computer vision techniques to extract information about the changing scene (e.g. hand gestures, face detection).

This chapter concentrates on how to generate webcam snaps using the JMF (Java Media Framework), and later chapters will apply various processing techniques to those pictures. The aim is to grab images as quickly as possible, and 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 implement suitable processing rates in later stages. Figure 1 shows the JMF application.

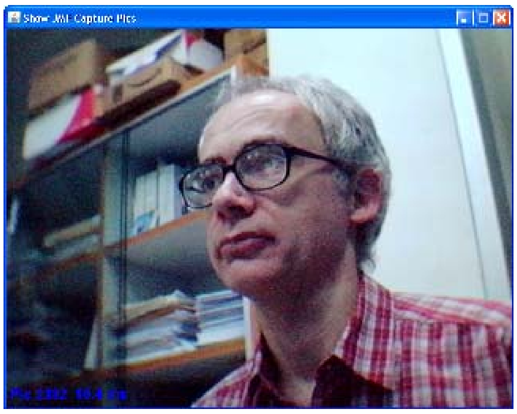


Figure 1. JMF Webcam Pictures.

The information about the picture is written in blue in the bottom-left corner of the image; in Figure 1 it says "Pic 2382 10.4 ms". The application always employs the video format with the largest frame size (normally 640 by 480 pixels), and the output shows that such a frame can be generates roughly every 10ms (on my slow test machine).

1. Displaying Pictures

The ShowJPics application uses the JMF Performance Pack for Windows v.2.1.1e (`http://java.sun.com/products/java-media/jmf/`) to grab pictures from the webcam, and display them in a JPanel. The class diagrams for the program, with only the public methods visible, are shown in Figure 2.

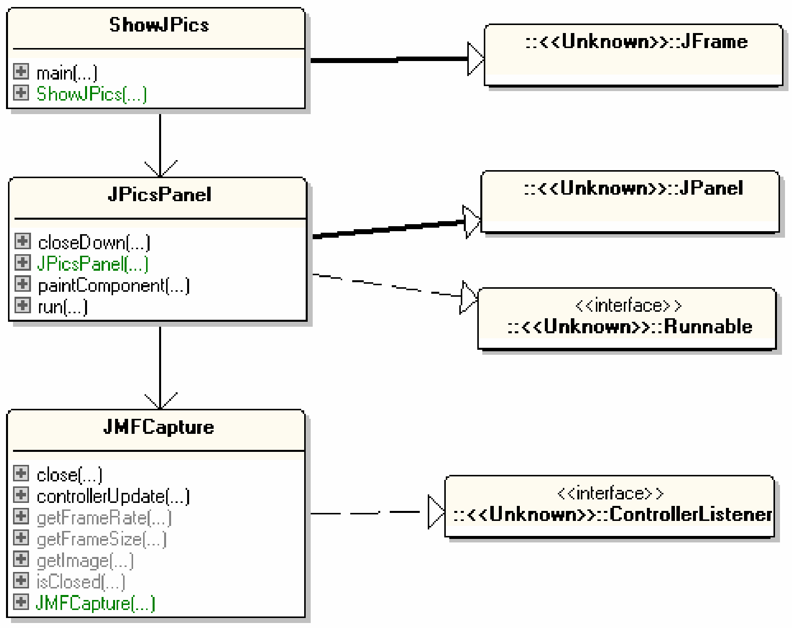


Figure 2. Class Diagrams for ShowJPics.

JPicsPanel is threaded so it can keep repeatedly calling getImage() in the JMFCapture object without causing the GUI parts of the panel to block.

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

1.1. Snapping a Picture Again and Again and ...

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

```
// globals
private static final int DELAY = 25;  //ms

private JMFCapture camera;
private BufferedImage image = null;
private JFrame top;
private volatile boolean isRunning;

// 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 JMFCapture(SIZE);

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

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

  while (isRunning) {
    long startTime = System.currentTimeMillis();
    im = camera.getImage();  // take a snap
    // im = scaleImage( camera.getImage(), SCALE_FACTOR);
                                    // take a snap, and resize
    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()
```

After the camera has been initialized, information about the frame size of the video source can be retrieved. This is used at the start of run() to modify the panel size, and adjust the top-level JFrame.

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 25 ms, which I chose by examining the statistics output to the screen when the program is executing. This makes the webcam 'movie' run at 40 frames/second (40 FPS) or slower, which is adequate for movie emulation.

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 can crash, and often causes Windows to start acting strangely. Consequently, all interactions with the device are localized inside run().

1.2. 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 JPicsPanel:

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

closeDown() sets isRunning to false, then waits for the camera to close. When isRunning is false, the loop in run() will eventually finish, and JMFCapture.close() will be called just before run() exits. close() sets a boolean to true inside JMFCapture, which allows JMFCapture.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 behavior is to put a call to JMFCapture.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 JMF source in the GUI thread rather than in the JPicsPanel thread.

1.3. Painting the Panel

The paintComponent() method draws the webcam picture in the panel, and writes the statistics on the bottom-left.

```
// globals
private DecimalFormat df;
private Font msgFont;


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

  int panelHeight = getHeight();
  g.setColor(Color.WHITE);    // white background
  g.fillRect(0, 0, getWidth(), panelHeight);

  // center the image
  int x = 0;
  int y = 0;
  if (image != null) {
    x = (int)(getWidth() - image.getWidth())/2;
    y = (int)(panelHeight - image.getHeight())/2;
  }
  g.drawImage(image, x, y, this);   // draw the snap

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

paintComponent() is called when the application is first made visible, which occurs before any images have been retrieved from the camera. In that case, paintComponent() draws the string "Loading..." at the bottom left of the panel (see Figure 3).

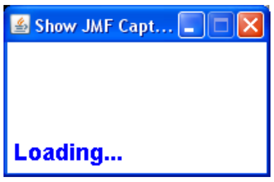


Figure 3. The Application During Loading.

The panel starts with a default size, which is updated once the video frame size is known.

Various diagnostics are also printed to the command window when the application is running.

1.4. Support Methods

JPicsPanel contains two support methods which aren't used in this application, but will be useful in later chapters: scaleImage() scales an image by a specified factor, while saveSnap() stores an image in a file.

```
// globals
private static final double SCALE_FACTOR = 0.5;  // for image

private static final String SAVE_FNM =
               System.getProperty("user.home") +
                                   "/desktop/jmfPic.jpg";
     // locate file on the Windows desktop (NOT portable)


private BufferedImage scaleImage(BufferedImage im, double scale)
```

```
// Make a scaled copy of im, assuming no alpha channel
{
  int scaledWidth = (int)((double)im.getWidth() * scale);
  int scaledHeight = (int)((double)im.getHeight() * scale);

  BufferedImage copy = new BufferedImage(scaledWidth, scaledHeight,
                                     BufferedImage.TYPE_INT_RGB);
  // create a graphics context
  Graphics2D g2d = copy.createGraphics();

  // resize a copy of the image
  g2d.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
                   RenderingHints.VALUE_INTERPOLATION_BILINEAR);
  g2d.drawImage(im, 0, 0, scaledWidth, scaledHeight, null);

  g2d.dispose();
  return copy;
}  // end of scaleImage()


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");  }
}  // end of saveSnap()
```

Examples of how to call these methods are commented out in the run() method shown above:

```
im = scaleImage( camera.getImage(), SCALE_FACTOR);
                                // take a snap, and resize

saveSnap(image, SAVE_FNM);
```

2. The Capture Device with JMF

Taking a snap with JMF can be summarized in seven steps:

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 a 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. Stage 6 is handled by the JMFCapture.getImage() method, described in section 3, and stage 7 (closing) is supported by the JMFCapture.close() method, explained in section 4.

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

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


public JMFCapture()
{
  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);
  }

  setToLargestVideoFrame(p);
  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 the JMF, then cycles through the list looking for the named device.

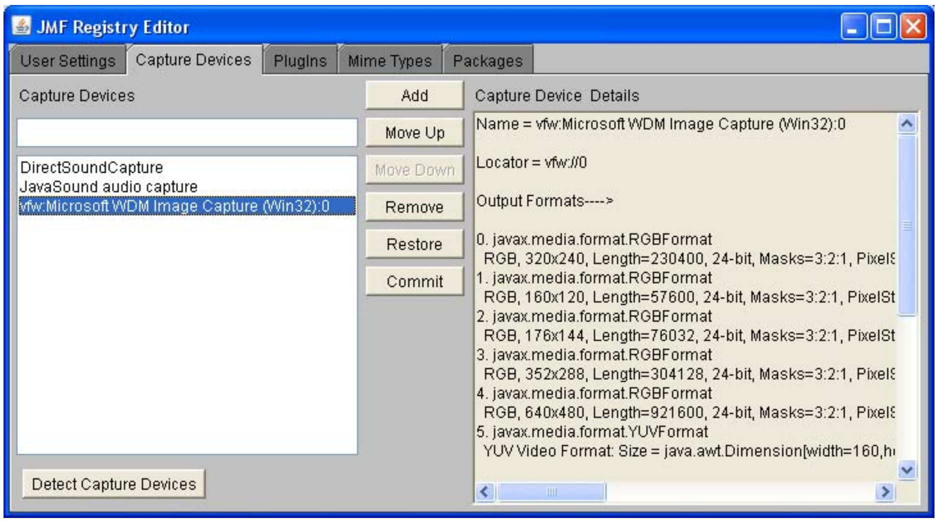
```
// globals
private static final String CAP_LOCATOR = "vfw://0";


private MediaLocator findMedia(String requireDeviceName)
// return a media locator for the specified capture device
{
  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);
  }

  CaptureDeviceInfo devInfo = null;
  int idx;
  for (idx = 0; idx < devices.size(); idx++) {
    devInfo = (CaptureDeviceInfo) devices.elementAt(idx);
    String devName = devInfo.getName();
    if (devName.equals(requireDeviceName))   // found device
      break;
  }

  MediaLocator ml = null;
  if (idx == devices.size()) {   // no device found with that name
    System.out.println("Device " + requireDeviceName + " not found");
    System.out.println("Using default media locator: " +
                                            CAP_LOCATOR);
    ml = new MediaLocator(CAP_LOCATOR);
  }
  else {   // found a suitable device
    System.out.println("Found device: " + requireDeviceName);
    storeLargestVf(devInfo);
    ml = devInfo.getLocator();   // this method may not work
  }

  return ml;
}  // 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. Its video properties are accessed by calling storeLargestVf() to record the largest frame size it can support.

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.

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).



```
DirectSoundCapture
JavaSound audio capture
vfw:Microsoft WDM Image Capture (Win32):0
```

```
Name = vfw:Microsoft WDM Image Capture (Win32):0

Locator = vfw://0

Output Formats---->

0. javax.media.format.RGBFormat
  RGB, 320x240, Length=230400, 24-bit, Masks=3:2:1, PixelS
1. javax.media.format.RGBFormat
  RGB, 160x120, Length=57600, 24-bit, Masks=3:2:1, PixelSt
2. javax.media.format.RGBFormat
  RGB, 176x144, Length=76032, 24-bit, Masks=3:2:1, PixelSt
3. javax.media.format.RGBFormat
  RGB, 352x288, Length=304128, 24-bit, Masks=3:2:1, PixelS
4. javax.media.format.RGBFormat
  RGB, 640x480, Length=921600, 24-bit, Masks=3:2:1, PixelS
5. javax.media.format.YUVFormat
  YUV Video Format: Size = java.awt.Dimension[width=160,h
```

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.

2.1. Using the Video with the Largest Frame Size

Figure 4 is a little hard to read, but the right-hand panel shows that the capture device can display video in six different sizes and/or encodings. By default, the first format is used, but it’s possible to change the selection, as illustrated by storeLargestVf() and setToLargestVideoFrame().

storeLargestVf() searches through the different video formats supported by the device, and stores the format with the largest frame size (typically 640 by 480 pixels) in largestVf.

```
// global
private VideoFormat largestVf = null;


private void storeLargestVf(CaptureDeviceInfo devInfo)
{
  Format[] forms = devInfo.getFormats();
  largestVf = null;
  double maxSize = -1;
  for (int i=0; i < forms.length; i++) {
    if (forms[i] instanceof VideoFormat) {
      VideoFormat vf = (VideoFormat) forms[i];
      Dimension dim = vf.getSize();
      double size = dim.getWidth() * dim.getHeight();
      if (size > maxSize) {
        largestVf = vf;
        maxSize = size;
      }
    }
  }
  if (largestVf == null)
    System.out.println("No video format found");
  else
    System.out.println("Largest format: " + largestVf);
}  // end of storeLargestVf()
```

After the player has been realized, setToLargestVideoFrame() employs largestVf to change the player's format.

```
private void setToLargestVideoFrame(Player player)
{
  FormatControl formatControl = (FormatControl) player.getControl(
                              "javax.media.control.FormatControl");
  if (formatControl == null) {
    System.out.println("No format controller found");
    return;
  }
  Format format = formatControl.setFormat(largestVf);
  if (format == null) {
    System.out.println("Could not change video format");
    return;
  }
  System.out.println("Video format changed to largest frame size");
}  // end of setToLargestVideoFrame()
```

Similar code can be used to access different attributes of the player's video format. For example, getFrameSize() returns the player's frame size:

```
// global
private Player p = null;

public Dimension getFrameSize()
{
  if (p == null)
    return null;
  FormatControl formatControl = (FormatControl) p.getControl(
                          "javax.media.control.FormatControl");
  if (formatControl == null)
    return null;
  VideoFormat vf = (VideoFormat) formatControl.getFormat();
  if (vf == null)
    return null;
  return vf.getSize();
}  // end of getFrameSize()
```

2.2. Waiting for a Player to Start

One of the difficult parts of JMF are the *eight* states that a player can move between. Thankfully, the JMFCapture code only needs to consider three: the realized, started, and closed states. After the player has been realized, it can be started with Player.start() (see JMFCapture()), but the code must wait until the player has transitioned to this state.

waitForStart() implements the blocking behavior by waiting for a wake-up signal for a waitSync object, and stores the outcome of the transition in a stateTransitionOK boolean.

```
// globals used for waiting until the player has started
private Object waitSync = new Object();
private boolean stateTransitionOK = true;


private boolean waitForStart()
// wait for the player to enter its Started state
{ synchronized (waitSync) {
    try {
      while (p.getState() != Controller.Started && stateTransitionOK)
        waitSync.wait();
    }
    catch (Exception e) {}
  }
  return stateTransitionOK;
} // end of waitForStart()
```

The wake-up signal is sent by my implementation of ControllerListener. controllerUpdate() which responds to various JMF events.

```
public void controllerUpdate(ControllerEvent evt)
// respond to events
{
  if (evt instanceof StartEvent) {   // the player has started
    synchronized (waitSync) {
      stateTransitionOK = true;
      waitSync.notifyAll();
    }
  }
  else if (evt instanceof ResourceUnavailableEvent) {
    synchronized (waitSync) {  // problem getting a player resource
      stateTransitionOK = false;
      waitSync.notifyAll();
    }
  }
} // end of controllerUpdate()
```

2.3. Waiting for a BufferToImage

I was surprised to discover that even after a player has ‘started’, JMF may still require several attempts to create a BufferToImage object. As a consequence, waitForBufferToImage() calls hasBufferToImage() repeatedly until it returns true; usually 2-3 tries are necessary.

```
// globals
private static final int MAX_TRIES = 7;
private static final int TRY_PERIOD = 1000;   // ms

private boolean closedDevice;


private void waitForBufferToImage()
```

```
{
  int tryCount = MAX_TRIES;
  System.out.println("Initializing BufferToImage...");
  while (tryCount > 0) {
    if (hasBufferToImage())   // initialization succeeded
      break;
    try {   // initialization failed so wait a while and try again
      System.out.println("Waiting...");
      Thread.sleep(TRY_PERIOD);
    }
    catch (InterruptedException e)
    {  System.out.println(e);  }
    tryCount--;
  }
  if (tryCount == 0) {
    System.out.println("Giving Up");
    System.exit(0);
  }

  closedDevice = false;   // device now available
}  // end of waitForBufferToImage()
```

hasBufferToImage() takes a snap, then checks if the resulting Buffer object really contains something.

```
// globals
private FrameGrabbingControl fg;
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);
  // 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.

3. Grabbing an Image

getImage() captures a snap as a Buffer object by calling FrameGrabbingControl.grabFrame(), then converts it to a BufferedImage.

```
// global
private FrameGrabbingControl fg;


synchronized public BufferedImage getImage()
{
  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 (BufferedImage) im;
}  // end of getImage()
```

4. Closing the Capture Device

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

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

JMFCapture.close() and JMFCapture.getImage() 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 JMFCapture.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.

5. More Information on JMF

JMF is aimed at time-based multimedia, not just image grabbing,. 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 book with several chapters on JMF 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`