Chapter NUI-2.5. Webcam Snaps Using VLC

The previous chapter was about using JavaCV to take webcam snaps. It does a fine job of processing the webcam's video input as a sequence of images, and is at the core of most of the examples in the rest of this book.

This chapter takes a slight detour to consider snap-taking without JavaCV and OpenCV. One historical reason for this is that some readers of earlier drafts complained about JavaCV's FrameGrabber containing memory leaks and crashing. I've had no such problems on my test machines running Windows XP and 7.

The main reason for exploring alternatives to JavaCV is simply to have more than one tool available for such a core feature as grabbing pictures.

I'm not going to consider the venerable Java Media Framework (JMF), due to its great age and lack of support for 64-bit versions of Windows. For readers who really want to use it, I refer you to my online chapter "Webcam Snaps Using JMF" at http://fivedots.coe.psu.ac.th/~ad/jg/nui01/.

In the past, I've recommended FMJ, an open-source project which is API-compatible with JMF (http://fmj-sf.net/). Unfortunately, that library is also starting to age, not having changed since 2007. Dust also seems to be settling upon Xuggler (http://www.xuggle.com/xuggler), which hasn't been updated since 2011. Readers interested in Xuggler should have a look at my online chapter about video watermarking at http://fivedots.coe.psu.ac.th/~ad/jg/javaArt7/.

This chapter is about reimplementing JavaCV's FrameGrabber using VLC (http://www.videolan.org/) and its Java binding, vlcj (http://code.google.com/p/vlcj/). The resulting class, VLCCapture, is shown in action in Figure 1.

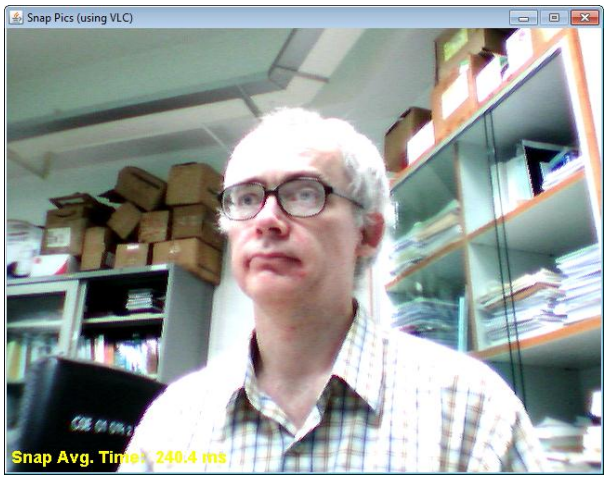


Figure 1. Webcam Pictures using VLC.

VLCCapture offers a similar interface to JavaCV's FrameGrabber, and so can be substituted for that class with a few lines of changes to the calling application.

© Andrew Davison 2013

1. VLC and vlcj

VLC is a popular open-source media player, with versions for Windows, Mac OS X, Linux, and many other platforms (http://www.videolan.org/). It comes pre-installed with support for just about every audio and video format, including streaming protocols. What might be less well known is its extensive API, offering programming access to video playback, streaming, conversion, and capture, with plentiful controls for adjusting video and audio input and output.

The API was initially written in C, but now comes with bindings for other languages, including vlcj for Java (http://code.google.com/p/vlcj/ and https://github.com/caprica/vlcj). There's a helpful VLC developer's wiki (http://wiki.videolan.org/Developers_Corner) and forum (http://forum.videolan.org/). The old vlcj website also has its own wiki (http://www.capricasoftware.co.uk/wiki/index.php?title=Main_Page) and examples (http://code.google.com/p/vlcj/wiki/SimpleExamples), although many are a bit out of date. This chapter's examples use vlcj v2.3.1, running over VLC v2.0.7.

2. Playing a Video

My Player.java example plays a specified video file using vlcj and VLC (see Figure 2).

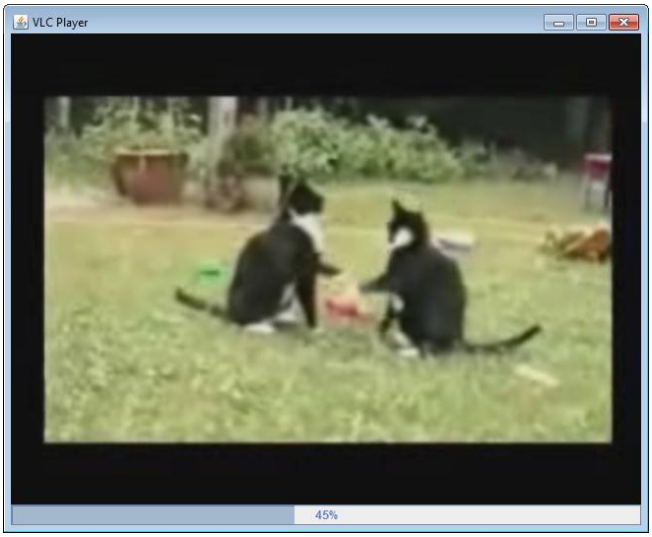


Figure 2. Playing a Video with vlcj.

The GUI includes a progress bar showing how much of the video has played. The bar can also be pressed to make the video jump to a particular position. The coding is based on a tutorial example at http://www.capricasoftware.co.uk/vlcj/tutorial2.php and on the "Minimal" and "Basic" applications at http://code.google.com/p/vlcj/wiki/SimpleExamples.

The Player() constructor contains the important video-related code:

```
// globals
private EmbeddedMediaPlayerComponent mPlayerComp;
```

 © Andrew Davison 2013

```
private MediaPlayer mPlayer;
private JProgressBar timeBar;


private Player(String fnm)
{
  super("VLC Player");

  // create video surface and media player
  mPlayerComp = new EmbeddedMediaPlayerComponent();

  Canvas canvas = mPlayerComp.getVideoSurface();
  canvas.setSize(640, 480);     // size of video surface

  mPlayer = mPlayerComp.getMediaPlayer();

  Container c = getContentPane();
  c.add(mPlayerComp, BorderLayout.CENTER);

  timeBar = new JProgressBar(0, 100);
  timeBar.setStringPainted(true);
  c.add(timeBar, BorderLayout.SOUTH);

  addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    { mPlayerComp.release();
      System.exit(0);
    }
  });

  pack();
  setLocationRelativeTo(null);  // center the window
  setVisible(true);

  // update the progress bar as the video progresses
  mPlayer.addMediaPlayerEventListener(new MediaPlayerEventAdapter()
    public void positionChanged(MediaPlayer mediaPlayer, float pos)
    { int value = Math.min(100, Math.round(pos * 100.0f));
      timeBar.setValue(value);
    }
  });

  // adjust the video position when the slider is pressed
  timeBar.addMouseListener( new MouseAdapter() {
    public void mousePressed(MouseEvent e)
    { float pos = ((float)e.getX())/timeBar.getWidth();
      mPlayer.setPosition(pos);
    }
  });

  System.out.println("Playing " + fnm + "...");
  mPlayer.playMedia(fnm);
}  // end of Player()
```

The application has three main vlcj-specific parts:

- the creation of a video surface and media player
- the loading and playing of the video
- the use of callbacks to monitor the video's progress

 © Andrew Davison 2013

The video surface and media player are created when an EmbeddedMediaPlayerComponent object is instantiated. EmbeddedMediaPlayerComponent extends Panel, so is also useful for building the GUI:

```
mPlayerComp = new EmbeddedMediaPlayerComponent();
```

The video surface is accessed with getVideoSurface() so its dimensions can be set:

```
Canvas canvas = mPlayerComp.getVideoSurface();
canvas.setSize(640, 480);   // size of video surface
```

The media player is referenced via the getMediaPlayer() method:

```
mPlayer = mPlayerComp.getMediaPlayer();
```

The video is started by calling MediaPlayer.playMedia():

```
mPlayer.playMedia(fnm);
```

This method returns immediately without waiting for the video to be loaded or start.

A video's progress can be monitored by setting up callbacks (listeners) using MediaPlayerEventListener (or its adapter, MediaPlayerEventAdapter). Player.java listens for playback position changes which trigger calls to MediaPlayerEventListener.positionChanged(). The new position is used to adjust the position of the progress bar.

The program also employs a mouse listener to detect presses on the progress bar, and MediaPlayer.setPosition() is called to move the video playback to a specific position.

More feature-rich media player examples can be found at http://code.google.com/p/vlcj/wiki/SimpleExamples.

3. Playing Input from a Webcam

Displaying input from a webcam utilizes similar ideas as those in Player.java, involving the creation of a video surface and a media player. However, several VLC arguments and options must be set in order to read input from a capture device instead of a file.

My CaptureTest application is shown running in Figure 3.

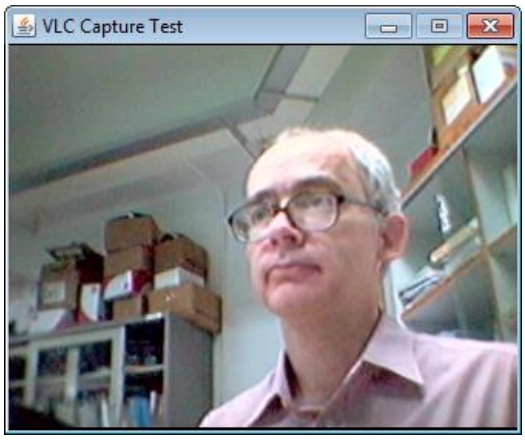


Figure 3. Displaying Webcam Input with CaptureTest.

 © Andrew Davison 2013

The CaptureTest() constructor creates a basic JFrame, leaving the VLC work to a playerPanel() method:

```
// global
private EmbeddedMediaPlayer mPlayer;


public CaptureTest()
{
  super("VLC Capture Test");
  Container c = getContentPane();
  c.add( playerPanel() );

  addWindowListener(new WindowAdapter() {
     public void windowClosing(WindowEvent e)
     { mPlayer.release();
       System.exit(0);
     }
  });

  pack();
  setResizable(false);
  setLocationRelativeTo(null);  // center the window
  setVisible(true);

  startPlayer();
}  // end of CaptureTest()
```

playerPanel() creates a media player, and a video surface attached to a JPanel:

```
// globals
private static final String[] VLC_ARGS = {
    "--no-audio",                // no audio decoding
    "--no-video-title-show",     // do not display title
    "--live-caching=50",         // reduce capture lag/latency
    "--quiet",                   // turn off VLC warnings
};

private EmbeddedMediaPlayer mPlayer;


private JPanel playerPanel()
{
  MediaPlayerFactory factory = new MediaPlayerFactory(VLC_ARGS);
  mPlayer = factory.newEmbeddedMediaPlayer();  // create media player

  Canvas canvas = new Canvas();
  canvas.setSize(640, 480);
  CanvasVideoSurface vidSurface = factory.newVideoSurface(canvas);
                                       // create video surface

  mPlayer.setVideoSurface(vidSurface);  // connect player and surface

  JPanel p = new JPanel();
  p.setLayout(new BorderLayout());
  p.add(canvas, BorderLayout.CENTER);   // add surface to a panel
  return p;
```

 © Andrew Davison 2013

```
}  // end of playerPanel()
```

In this example, I utilize a MediaPlayerFactory class to create the media player and video surface, instead of EmbeddedMediaPlayerComponent. The MediaPlayerFactory constructor can take a string of VLC command line arguments, to modify the behavior of the resulting player and/or surface. There are a lot of possible arguments, most of which are described at http://wiki.videolan.org/VLC_command-line_help (or you can type vlc –H at the command line to get a shorter list).

Starting the player is also more complicated than previously:

```
// globals
private static final String CAP_DEVICE = "dshow://";   // for Windows

private static final String CAMERA_NAME =
                      "USB2.0 Camera";  // webcam name

private EmbeddedMediaPlayer mPlayer;

private void startPlayer()
{
  String[] options = {
    ":dshow-vdev=" + CAMERA_NAME,
    ":dshow-size=640x480",
    ":dshow-adev=none",     // no audio device being used
  };
  mPlayer.playMedia(CAP_DEVICE, options);
}  // end of startPlayer()
```

The MediaPlayer.playMedia() method requires the capture device's address, which is "dshow://" on Windows (short for DirectShow), with additional options to specify the device name and capture size. These options are written in a vlcj format, which starts with ":" rather than "--".

The easiest way of finding a video device's name is by starting the VLC media player application. Lists of video and audio device names can be found in the "Open Media" dialog box displayed by the menu item Media >> "Open Capture Device…" (see Figure 4).

 © Andrew Davison 2013

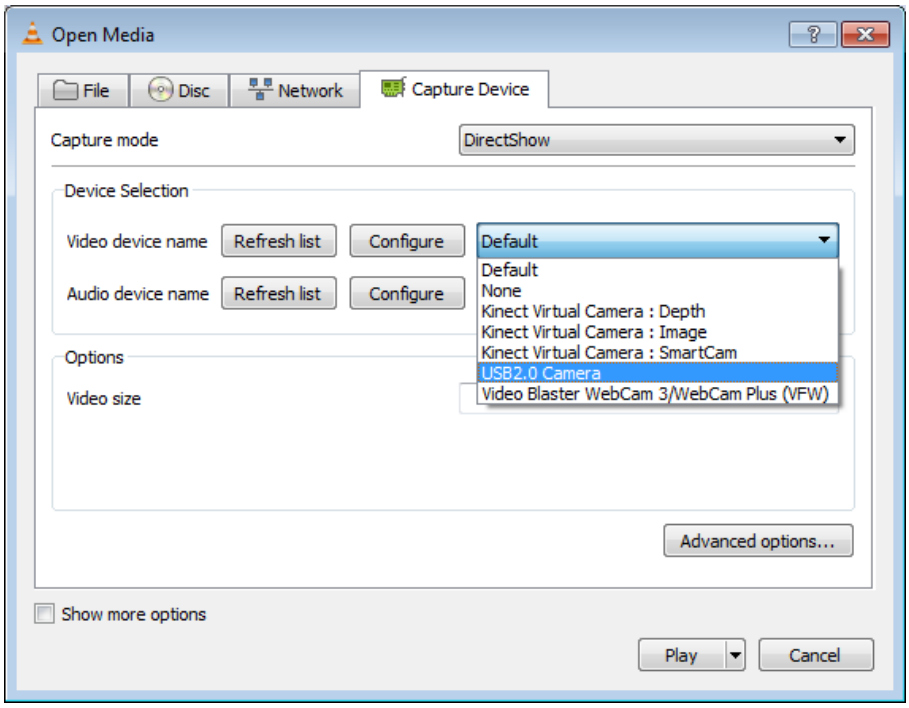


Figure 4. Device Names in VLC.

Once the video and audio device names are selected, the "Play" button will trigger the display of input from the capture device (Figure 5).

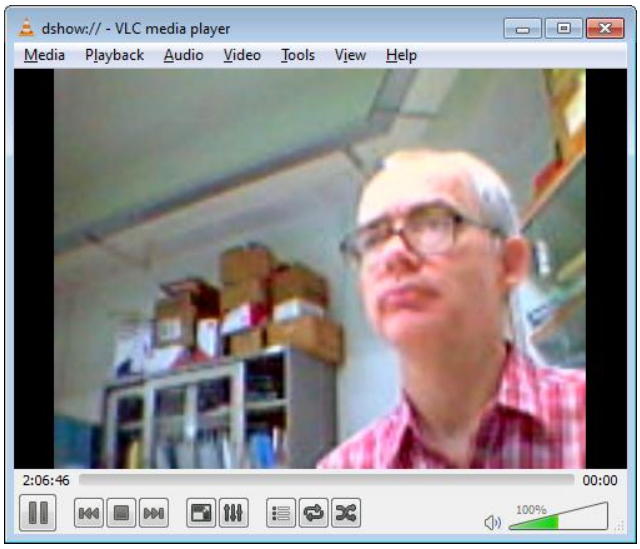


Figure 5. Webcam Capture in the VLC Media Player.

This is a good way of confirming that VLC is working independently of your vlcj code.

The capture device address used in MediaPlayer.playMedia() will vary depending on the video protocol. For example, on Windows it is "dshow://" (short for DirectShow), "v4l2:// or perhaps v4l:// for Linux (for a video4linux device), and "qtcapture://" for Mac OS X.

 © Andrew Davison 2013

If you move your code to a different platform, the MediaPlayer.playMedia() options that mention the device protocol will also need to change. For example, the DirectShow options "dshow-vdev", "dshow-size", and "dshow-adev" in the startPlayer() method above will have to change.

A simple way discovering options names is via VLC's "Open Media" dialog box. When its "Show more options" tick box is selected, useful additional text fields appear (see Figure 6).

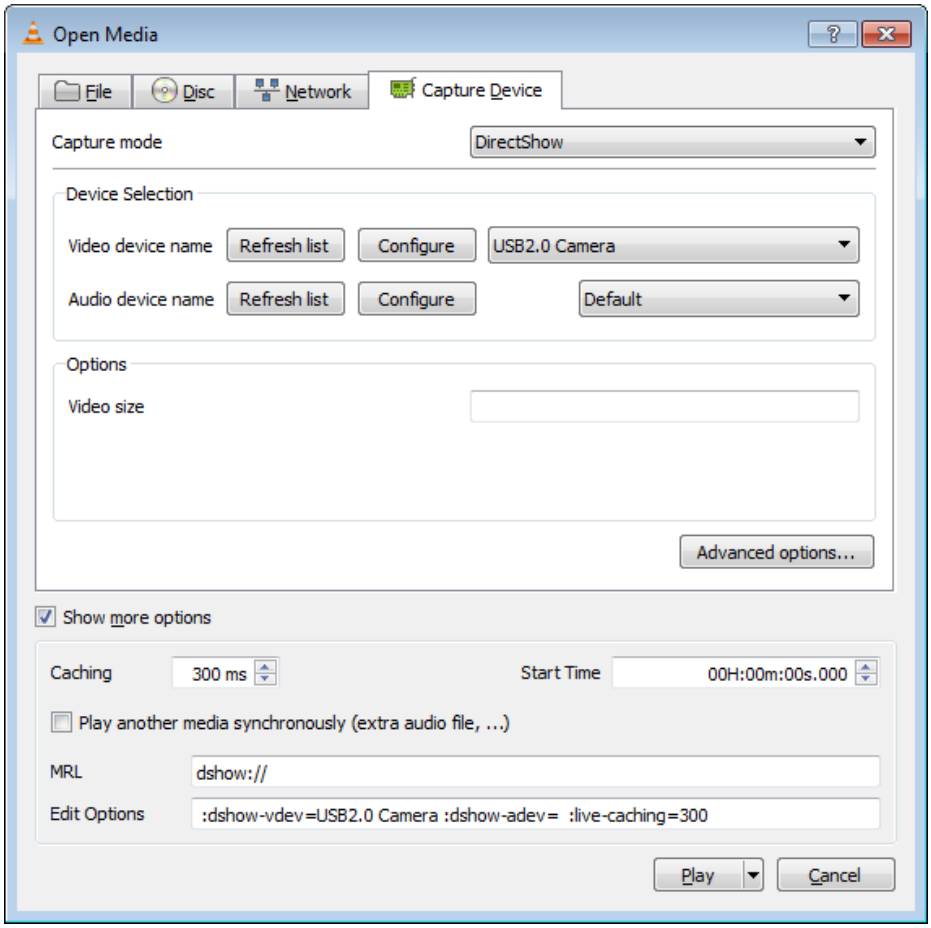


Figure 6. Device Names and Options in VLC.

The capture device address is shown in the "MRL" field, and the "Edit-Options" field contains values for "dshow-vdev" and "dshow-adev". The names of these options will vary depending on the device protocol being used.

4. Snapping Images with the Capture Device

The CaptureTest.java example from the previous section renders the capture device's video stream onto a canvas, but what I actually need for my applications is a sequence of images (or snaps) pulled from that stream. The Capturer.java application in this section shows how to do that, extracting images from the video stream, and rendering them onto a JPanel. The result is shown in Figure 7, and looks just like the CaptureTest example in Figure 3.

 © Andrew Davison 2013



Figure 7. Displaying Webcam Input with Capturer.

Although the screenshots in Figures 3 and 7 look the same, there is a noticeable difference between the two programs when running. The image-snapping Capturer example of this section updates the panel more slowly since VLC is only able to generate an image once every 200 ms (at least on my test machines), equal to about 5 FPS. This is quite a bit slower than the 10 FPS possible with JavaCV.

The UML class diagrams for Capturer are given in Figure 8.

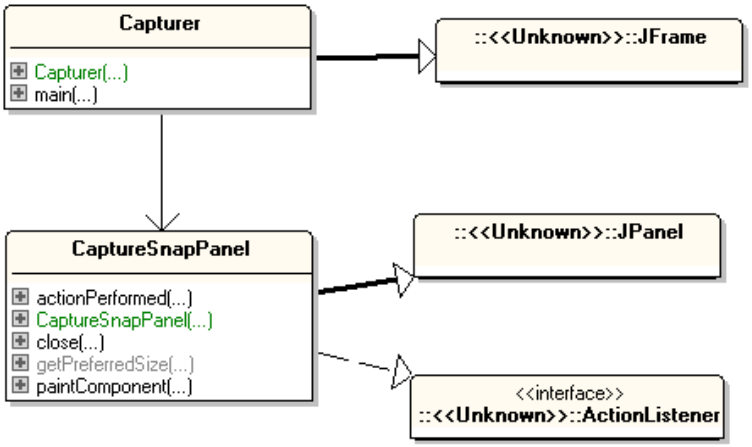


Figure 8. Class Diagrams for Capturer.

The interesting VLC parts of the code are in the CaptureSnapPanel class. The constructor creates a 'headless' media player, without a video surface. A surface isn't needed since the panel will be displaying the snapped images not the video itself.

```
// globals
// dimensions of the panel and snapped images
private static final int P_WIDTH = 640;
private static final int P_HEIGHT = 480;

private static final int SNAP_INTERVAL = 100;   // ms
    // rate at which snaps are taken and displayed in the panel

private static final String CAP_DEVICE = "dshow://";
```

 © Andrew Davison 2013

```
private static final String CAMERA_NAME =
                                  "USB2.0 Camera";

private static final String[] VLC_ARGS = {
  "--intf", "dummy",              // no interface
  "--vout", "dummy",              // no video output
  "--no-audio",                   // no audio decoding
  "--no-video-title-show",        // do not display title
  "--no-stats",                   // no stats
  "--no-sub-autodetect-file",     // no subtitles
  "--no-snapshot-preview",        // no snapshot preview
  "--live-caching=50",            // reduce capture lag/latency
  "--quiet",                      // turn off VLC warnings
};

private MediaPlayer mPlayer;
private Timer timer;     // for updating the current image


public CaptureSnapPanel()
{
  MediaPlayerFactory factory = new MediaPlayerFactory(VLC_ARGS);
  mPlayer = factory.newHeadlessMediaPlayer();

  String[] options = {
    ":dshow-vdev=" + CAMERA_NAME,
    ":dshow-size=" + P_WIDTH+"x"+P_HEIGHT,
    ":dshow-adev=none",    // no audio device required
  };
  mPlayer.startMedia(CAP_DEVICE, options);

  // update the current image every SNAP_INTERVAL ms
  timer = new Timer(SNAP_INTERVAL, this);
  timer.start();
} // end of CaptureSnapPanel() constructor
```

According to the vlcj documentation, VLC may still spawn a display for a headless media player, and so the MediaPlayerFactory object must be initialized with numerous VLC arguments to disable any such display.

The video is started with a call to MediaPlayer.startMedia(), which waits for the video to start before returning (earlier I used playMedia() which returns immediately).

The constructor finishes with the start of a timer, which triggers a call to actionPerformed() every SNAP_INTERVAL milliseconds:

```
// globals
private MediaPlayer mPlayer;
private BufferedImage snapIm;

public void actionPerformed(ActionEvent e)
// called by the timer to take a snapshot
{
  snapIm = mPlayer.getSnapshot(P_WIDTH, 0);
            // 0 means that video's aspect ratio is utilized
  if (snapIm == null)
    return;

  if (snapIm.getWidth() > 0)
```

 © Andrew Davison 2013

```
    repaint();
}  // end of actionPerformed()
```

MediaPlayer.getSnapshot() returns an image of the required width, while utilizing the aspect ratio of the video. If the image contains some data then the panel is redrawn by paintComponent():

```
public void paintComponent(Graphics g)
// put the current snapshot in the middle of the panel
{
  super.paintComponent(g);   // repaint standard stuff first
  if (snapIm != null)
    g.drawImage(snapIm, (P_WIDTH-snapIm.getWidth())/2,
                        (P_HEIGHT-snapIm.getHeight())/2, null);
}
```

Both actionPerformed() and paintComponent() contain tests to check if the image is null, which often occurs at the start of Capturer's execution. This is true even when the video is started using MediaPlayer.startMedia(), which waits until the video is playing before returning. On my slow XP test machine, this behavior meant that the JPanel sometimes remained blank for a second before the first snap appeared.

5. A Replacement for FrameGrabber

I've now developed enough code to replace JavaCV's FrameGrabber with VLCCapture, which employs VLC internally while offering a grabber-like public interface. This means that my test-rig for VLCCapture (the classes SnapPics and PicsPanel) are virtually unchanged from the code in chapter NUI-2 (http://fivedots.coe.psu.ac.th/~ad/jg/nui02/). The program is shown running back in Figure 1, and its class diagrams appear in Figure 9.

 © Andrew Davison 2013

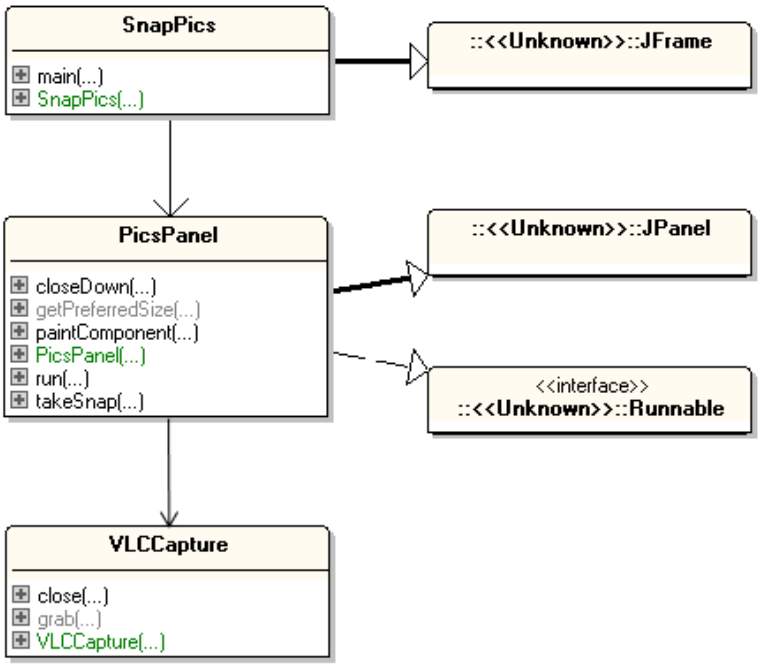


Figure 9. Class Diagrams for SnapPics.

PicsPanel is threaded so it can repeatedly call grab() in the VLCCapture object without causing the GUI parts of the panel to block.

As in the JavaCV version, SnapPics closes down PicsPanel when its close box is clicked. Also, if the user presses numpad 5, enter or space, then a grayscale version of the current image is saved in a pics/ directory.

5.1. Snapping a Picture Again and Again and ...

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

```
// globals
private static final int WIDTH = 640;
private static final int HEIGHT = 480;
private static final int DELAY = 100;  // ms

private static final String CAMERA_NAME = "USB2.0 Camera";

private BufferedImage snapIm = null;
private volatile boolean isRunning;
private volatile boolean takeSnap = false;

// 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 */
```

 © Andrew Davison 2013

```
{
  VLCCapture grabber = new VLCCapture(CAMERA_NAME, WIDTH, HEIGHT);
  if (grabber == null)
    return;

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

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

    snapIm = grabber.grab();  // take a snap

    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.close();    // close down the grabber
}  // end of run()
```

The VLCCapture camera is initialized using the name of the device, and the required image dimensions.

The DELAY value used in run() is 100 ms, but I'm being overly optimistic. The statistics output to the screen when the program is executing (as shown in Figure 1) indicate that a VLC snap takes about 200 ms to be processed, or runs at 5 FPS, which is quite a bit slower than JavaCV.

5.2. Camera Capture with VLC

The VLCCapture constructor starts in much the same as the CaptureSnapPanel class from earlier, but is passed the name of the camera and the image dimensions.

```
// globals
private static final String CAP_DEVICE = "dshow://";

private static final String[] VLC_ARGS = {
  "--intf", "dummy",              // no interface
  "--vout", "dummy",              // no video output
  "--no-audio",                   // no audio decoding
  "--no-video-title-show",        // do not display title
  "--no-stats",                   // no stats
  "--no-sub-autodetect-file",     // no subtitles
```

 © Andrew Davison 2013

```
  "--no-snapshot-preview",        // no snapshot previews
  "--live-caching=50",            // reduce capture lag/latency
  "--quiet",                      // turn off warnings
};

private MediaPlayerFactory factory = null;
private MediaPlayer mPlayer = null;
private String cameraName;
private int imWidth;


public VLCCapture(String camName, int width, int height)
{
  factory = new MediaPlayerFactory(VLC_ARGS);
  mPlayer = factory.newHeadlessMediaPlayer();
  cameraName = camName;
  imWidth = width;

  System.out.println("Initializing grabber for \"" +
                                     cameraName + "\" ...");
  String[] options = {
    ":dshow-vdev=" + cameraName,
    ":dshow-size=" + width+"x"+height,
    ":dshow-adev=none",    // no audio device required
  };
  mPlayer.startMedia(CAP_DEVICE, options);
} // end of VLCCapture() constructor
```

grab() wraps up a call to MediaPlayer.getSnapshot() and tests the result:

```
public BufferedImage grab()
{
  BufferedImage im = mPlayer.getSnapshot(imWidth, 0);
      // get image width x ?? with aspect ratio maintained

  if ((im == null) || (im.getWidth() == 0)) {
    System.out.println("No snap available");
    return null;
  }
  return im;
}  // end of grab()
```

close() terminates the media player and factory:

```
// globals
private MediaPlayerFactory factory;
private MediaPlayer mPlayer;


public void close()
{
  System.out.println("Closing grabber for \"" +
                               cameraName + "\" ...");
  if(mPlayer != null) {
    mPlayer.release();
    mPlayer = null;
  }
```

 © Andrew Davison 2013

```
  if(factory != null) {
    factory.release();
    factory = null;
}  // end of close()
```

One significant difference between VLCCapture and JavaCV's FrameGrabber is that my code is hardwired to look for a DirectShow device. This is apparent in the CAP_DEVICE string value, and some of the arguments in the options string in the constructor. To be clear, this is a limitation of my coding, not of vlcj and VLC, which support a range of platforms. For example. CAP_DEVICE would be "v4l2://" for Linux and "qtcapture://" for Mac OS X.

Another difference is that the VLCCapture constructor takes a camera name rather than an ID number. This means that a user cannot just supply an ID of 0 and hope for the best. The name of the device can be discovered either by using the VLC "Open Media" dialog box (e.g. see Figure 6) or the command line tools discussed in the previous chapter (i.e. CommandCam, DevCon, and FFmpeg).

 © Andrew Davison 2013