Kinect Chapter 11. Kinect Capture

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

This book grew out of a larger writing project on Natural User Interfaces (NUIs). A NUI aims to simplify the user's interaction with the PC, by replacing the mouse and keyboard by visual and audio forms of communications (i.e. more *natural* interaction).

The webcam is an important 'building block' for creating novel input mechanisms. The camera repeatedly delivers pictures (e.g. of the user's hand or face) to a processing stage, which employs image manipulation and computer vision techniques to extract information about the scene (e.g. hand gestures, face detection).

My NUI writing cover topics such as motion detection, color blob tracking, face identification and recognition, augmented reality, and tangible user interfaces. You can find the code, and draft versions of the chapters, online at http://fivedots.coe.psu.ac.th/~ad/jg/ in the "Natural User Interfaces" section. The examples all utilize a webcam input class called JMFCapture which retrieves video frames as sequences of BufferedImages.

The aim of this chapter is to reimplement JMFCapture to use the Kinect camera as its input source. This renamed KinectCapture class has almost the same interface as JMFCapture, allowing it to be used as a "drop-in" replacement for JMFCapture in the NUI examples at http://fivedots.coe.psu.ac.th/~ad/jg/

To prove the point, the next chapter will look at using the Kinect as an input source for OpenCV (the popular computer vision library). I'll utilize OpenCV (actually, its Java binding called JavaCV) to implement a simple kind of motion detection. The algorithms (and code) in that chapter are identical to the online NUI chapter on motion detection at http://fivedots.coe.psu.ac.th/~ad/jg/nui03/. Only a few lines have been changed so a camera variable refers to a KinectCapture object rather than a JMFCapture instance.

The focus of this chapter is on implementing KinectCapture, together with a small test-rig to show how to grab images 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 CapturePics application.

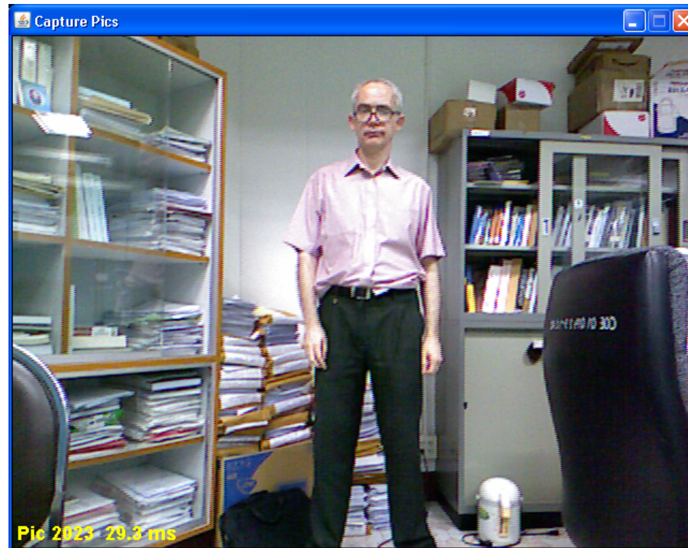

Figure 1. The CapturePics Application

Information about the picture is written in yellow in the bottom-left corner of the image; in Figure 1 it says "Pic 2023 29.3 ms", which indicates that the Kinect camera is working at a frame rate of about 1000/29.3 == 34 FPS.

The frame size is normally 640 by 480 pixels, the normal resolution of the Kinect's camera, but it's also possible to switch to high resolution (1280 x 1024), but the frame rate drops to around 7 FPS. High-res can be useful for more complicated image processing such as face recognition.

1. Displaying Pictures

Class diagrams for the CapturePics application appear in Figure 2.

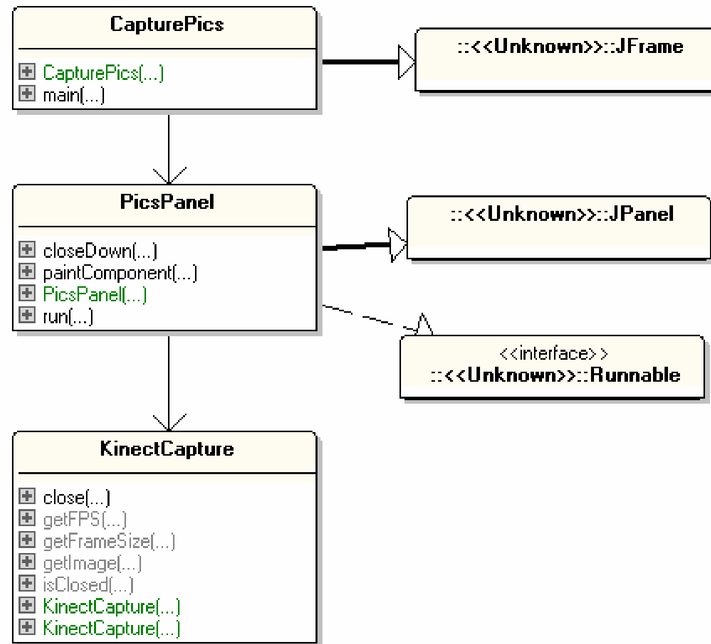

Figure 2. Class Diagrams for CapturePics.

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

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

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

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

```
// globals
private static final int DELAY = 35;   //ms

private KinectCapture 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 KinectCapture();     // normal resolution
  // camera = new KinectCapture(Resolution.HIGH);
```

```
  // 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
    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) {}
    }
  }

  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 35 ms, which I chose by examining the statistics output to the screen when the program is executing (as shown in Figure 1). This makes the Kinect 'movie' run at about 30 FPS or slower.

The method includes two examples of how to initialize the camera (one of them commented out):

```
camera = new KinectCapture();    // normal resolution
// camera = new KinectCapture(Resolution.HIGH);
```

The no-argument call creates a normal-size image (using a resolution of 640 x 480), while the Resolution.HIGH argument sets the size to 1280 x 1024.

1.2. Terminating the Application

When the user presses the close box in the JFrame, closeDown() is called in PicsPanel:

```
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 KinectCapture.close() will be called just before run() exits. close() releases the Kinect's context, and after that closedDown() will return true, permitting the application to terminate.

This approach means that the program's GUI may 'freeze' for a short time when the close box is clicked, since closeDown() blocks the GUI thread while it waits.

1.3. Painting the Panel

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

```
// globals
private BufferedImage image = null;

private int imageCount = 0;
private long totalTime = 0;
private DecimalFormat df;
private Font msgFont;


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

  int panelHeight = getHeight();
  g.setColor(Color.GRAY);    // gray 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.YELLOW);
  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 at a default size, which is updated once the video frame size is known.

2. Capturing the Kinect's Camera

The KinectCapture constructor handles a resolution setting (Resolution.NORMAL or Resolution.HIGH) by passing it to configOpenNI().

```
public KinectCapture()
{  this(Resolution.NORMAL);  }

public KinectCapture(Resolution res)
{  configOpenNI(res); }
```

configOpenNI() utilizes an OpenNI context to create an ImageGenerator:

```
// globals
private Context context;
private ImageGenerator imageGen;
```

```
private boolean isReleased;
private int imWidth, imHeight;
private int fps;


private void configOpenNI(Resolution res)
{
  try {
    context = new Context();

    // add the NITE Licence
    License licence = new License("PrimeSense",
                          "0KOIk2JeIBYClPWVnMoRKn5cdY4=");
    context.addLicense(licence);

    imageGen = ImageGenerator.create(context);

    MapOutputMode mapMode = null;
    if (res == Resolution.HIGH)    // set xRes, yRes, FPS
      mapMode = new MapOutputMode(1280, 1024, 15);
    else   // default to NORMAL
      mapMode = new MapOutputMode(640, 480, 30);

    imageGen.setMapOutputMode(mapMode);
    imageGen.setPixelFormat(PixelFormat.RGB24);

    // set Mirror mode for all
    context.setGlobalMirror(true);

    context.startGeneratingAll();
    System.out.println("Started context generating...");
    isReleased = false;

    ImageMetaData imageMD = imageGen.getMetaData();
    imWidth = imageMD.getFullXRes();
    imHeight = imageMD.getFullYRes();
    fps = imageMD.getFPS();
    System.out.println("(w,h); fps: (" + imWidth + ", " +
                                 imHeight + "); " + fps);
  }
  catch (Exception e) {
    System.out.println(e);
    System.exit(1);
  }
}  // end of configOpenNI()
```

The Resolution values are used to choose between two calls to MapOutputMode which set different frame sizes and rates.

2.1. Obtaining an Image

getImage() waits for the ImageGenerator to be updated and then returns a BufferedImage version of its current image.

```
// globals
private BufferedImage image;
private boolean isReleased;
private Context context;
```

```
private ImageGenerator imageGen;


public BufferedImage getImage()
{
  if (isReleased)
    return null;
  try {
    context.waitOneUpdateAll(imageGen);
    ByteBuffer imageBB = imageGen.getImageMap().createByteBuffer();
    return bufToImage(imageBB);
  }
  catch (GeneralException e)
  {  System.out.println(e); }
  return null;
}  // end of getImage()
```

The isReleased boolean is true if the OpenNI context has been released, and so the method returns null in that case.

2.2. Closing Down

The close() method stops the context, and sets isReleased to true.

```
// globals
private boolean isReleased;
private Context context;


public void close()
{
  try {
    context.stopGeneratingAll();
  }
  catch (StatusException e) {}
  context.release();
  isReleased = true;
} // end of close()
```

A user of KinectCapture should call isClosed() to check if the Kinect has been shutdown, which is only the case when isReleased is true:

```
public boolean isClosed()
{  return isReleased;  }
```