EECS498 Autonomous Robotics Laboratory
Winter 2012

ArmLab, part 1

OBJECTIVE. The goal of this lab is to familiarize you with some of the software tools provided by
the staff: jcam and vis. To do this, we will implement a simple computer vision application capable of
detecting an LED and providing a graphical display that overlays a rectangle over the LED.

DELIVERABLES. This lab contains a number of TASKS that require your response. In your lab
write up, you should restate the question (you can copy and paste from this document) and then provide
your answer. In short, your writeup should be comprehensible without needing to reference this
document.

SUBMITTING YOUR WRITEUP. Your lab writeup must be emailed in the following format:

TO: eecs498-staff@april.eecs.umich.edu
SUBJECT: ArmLab Part 1: <uniqname1>, <uniqname2>, <uniqname3>
CC: <all of your team members>

Your email should contain exactly two attachments:

1. A PDF of your writeup. Equations must be typeset, and figures must be rendered by april.vis.
   No other formats (including .doc) are acceptable. Latex is always appreciated, but not required.
   note: Your PDF must be less than 5MB in size. This is generally easily achievable by using
   images of sensible resolution. If this is impossible, put your PDF on a web server and include in
   your email 1) a URL to the file and 2) the md5sum of the PDF.

2. A photo of your collaboration certification, signed by all members. It may be hand-written,
   and be no more than 200kB in size.

Deviations from this format will be penalized. If you receive a "moderation pending" message, it is
harmless.

TEAMWORK. As a reminder, our class collaboration policy requires your team to work together on
each problem. You may not divide up the work and proceed in parallel.
Cameras

Cameras are specified in jcam by using a URL. These URLs specify a type of image source (for example, a camera that implements the dc1394 protocol) and an identifier for the particular camera. For example, the URL of the camera on my desk is "dc1394://b09d0100af0611".

The primary tool for experimenting with jcam is JCamView. It will show all available cameras and allow you to change the settings associated with them. You can invoke it with:

    bash% java april.jcam.JCamView

A single camera typically offers multiple formats of data. A format is comprised of a resolution and a pixel format. For example, "720 x 480 (BAYER_RGGB)" is a camera format in which the pixels are organized in a Bayer pattern.

Note that you can easily save camera images to disk; right click on the image and select "Save Image".

**TASK 1.** The aperture controls how much light enters the lens, but it also has an effect on the depth of field. By adjusting the aperture, take two pictures (of the same scene), showing how the depth of field is affected by aperture.

**TASK 2.** The easiest way to spot lens distortion is to look for edges in the image which appear curved but are in reality straight. Lens distortion can vary based on focal length. Does your lens exhibit more lens distortion at long focal lengths or at short focal lengths? Take an image at both lengths and describe the distortion: which is worse?

**TASK 3.** Your camera is a Bayer pattern based color camera. Select the GRAY8 mode on your camera and zoom in so that you can see individual pixels. What do you see (include an image)? Why does it look like this?

**TASK 4.** Select the camera mode "BAYER_RGGB" (or similar). In this mode, JCam is applying a simple debayering algorithm for you. This process can result in chromatic artifacts, as described in lecture. How can you construct a scene so that these chromatic effects are very noticeable? Show an image (including a zoom) showing these artifacts.

**JCam API**

The steps necessary to acquire images using JCam are listed below. See april.jcam.LatencyTest for a relatively simple example.

1. Create an Image Source by using the factory method "isrc = ImageSource.make(url)". You can set the format programatically, or you can include format information in the URL (try pressing the "Print camera URL" button in JCamView.)

2. Call isrc.start(). This initiates the streaming of data from the camera. Note that you cannot change any camera parameters once you call start().

3. Call isrc.getFrame(). Note that this function returns raw data from the camera, and is is not necessarily easy-to-use; depending on the selected camera format, it could be a simple gray
scale image or a motion-JPEG compressed image.

4. Call `ImageConvert.convertToImage()` to convert the camera-specific format to a BufferedImage (Java's "standard" image format). Note that you'll need the current ImageSourceFormat, which can be obtained via `isrc.getCurrentFormat()`.

5. Do something with the BufferedImage.

6. Go to step 3.

**TASK 5.** Write a Java class whose main method acquires frames from the camera, printing out a dot to the terminal every time a frame is acquired. Measure the frame rate (see `System.nanoTime()`). What frame rate did you measure? Do you get different frame rates for different camera formats? Speculate why the camera format would affect the frame rate.

**Visualization with Vis**

We now modify our program to display the current image in Vis-- in essence, we're creating a simple version of JCamView. To do this, you'll need to create a Java Swing window (a "JFrame") and add a VisCanvas to it. A standard sequence looks like this (see `april.vis.examples.Zoo` for a not-too-complicated example):

```java
JFrame jf = new JFrame("My Example");
jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
jf.setSize(600,400);
jf.setLayout(new BorderLayout());

VisWorld vw = new VisWorld();
VisLayer vl = new VisLayer(vw);
VisCanvas vc = new VisCanvas(vl);

jf.add(vc);
jf.setSize(600, 400);
jf.setVisible(true);
```

Make sure you understand what's going on at each step. To display the image acquired from jcam, you need to add an image object to a VisWorld.Buffer. In short, every time you get a new frame from the camera, you can do something like this:

```java
VisWorld.Buffer vb = vw.getBuffer("image");
vb.addBack(new VzImage(theimage));
vb.swap();
```

**TASK 6.** Complete the tasks above, and include a screen shot of the output. Why is your image upside-down? (Hint: consider the standard coordinate system used by images versus the coordinate system used by OpenGL). Fix the problem by using the `VisChain` object and introducing a number of rigid-body transformations before your object. I.e., change the line above to something like:

```java
vb.addBack(new VisChain(<stuff>, new VzImage(theimage));
```
You can create simple rigid-body transformations using functions like LinAlg.translate, LinAlg.scale, LinAlg.rotateX, etc.

Include the code you used to fix your images.

**TASK 7.** Let's overlay a square on the center of our image. Use a different VisWorld.Buffer named "square" and add a VzSquare. You will again need a VisChain to position and scale the square so that it is 10 pixels by 10 pixels. Include a screen shot.

**LED Tracking**

We're now ready to search for an LED in the image. By looking for the brightest spot in each frame and putting a box around it, we'll show that we can track the LED. There are a couple ways of accessing the data in an image. The first (see the java.awt.image.BufferedImage documentation) is to simply call:

```java
int rgb = theimage.getRGB(x, y);
```

This will pack the red, green, and blue values into a single integer value like this:

```java
rgb = (r << 16) | (g << 8) | (b);
```

Naturally, you can iterate over the whole image using two for loops:

```java
for (int x = 0; x < theimage.getWidth(); x++)
    for (int y = 0; y < theimage.getHeight(); y++)
        ...
```

You can also set pixels using theimage.setRGB(x, y, rgb).

**TASK 8.** Implement a loop that iterates over every pixel, extracts the individual r, g, and b components (watch out for sign extension!), multiplies each component by a scale factor (scaler, scaleg, scaleb), then reassembles the rgb integer and stuffs it back into the image.

**TASK 9.** Using the april.util.ParameterGUI class, add sliders to the bottom of your JFrame (using BorderLayout.SOUTH) to control the three scale factors from task 8. As you slide the sliders, the image should adjust dynamically. Set the range of these sliders to be from zero to 256.

(Hint: see pg.addDouble(), and pg.gd(). For this task, you do not need to implement ParameterListener, but you may wish to take a look at it--- it will be useful to you in the future.).

**TASK 10.** Set the red and green sliders to zero. What happens when you set the blue slider to 256? Explain.

**TASK 11.** Find the brightest pixel in the image and adjust your code to draw the box around it. How did you determine the "brightest pixel"? Show a screen shot.

**TASK 12.** Another way of accessing pixels in a BufferedImage is to retrieve the raster using a call like:

```java
int data[] = ((DataBufferInt) (im.getRaster().getDataBuffer())).getData();
```
The data object points to the underlying memory that the BufferedImage uses internally. Accessing pixels directly, rather than through getRGB and setRGB, can often be much faster. All of the pixels are present in this one-dimensional array. To access pixel (x,y), what index should you use?

Using System.nanoTime(), compare the speed difference between using getRGB and using the raster. Note that because Java uses a garbage collector which can cause timing variations, you may want to run your experiment a few hundred times and compute a reasonable statistic.

Next, try changing the order of your for loops from

```java
for (int x = 0; x < theimage.getWidth(); x++)
    for (int y = 0; y < theimage.getHeight(); y++)
```

to

```java
for (int y = 0; y < theimage.getHeight(); y++)
    for (int x = 0; x < theimage.getWidth(); x++)
```

You should now have speed measurements for all four variations (getRGB, raster) and (x then y, y then x).

1) Describe how you made your timing measurements robust (how many trials? what statistic did you use? How do you know that this worked?)

2) Produce a visualization of your timing data (the particular type of visualization is up to you, but you'll be evaluated on the effectiveness of your choice.)

3) Explain the timing variations.

**TASK 13.** The "brightest spot" caused by the LED is probably actually a big blob of pixels. Devise a way to find the middle of this blob. An approximate method is fine, but explain your reasoning.

Submit both your actual code and an explanatory pseudo-code. Along with the pseudo-code, include a paragraph of explanatory prose.

**CHECKOFF.** Demo your TASK 13 system to a staff member.