Massachusetts Institute of Technology

Robotics: Science and Systems I

Lab 4: The Carmen Robot Control Package and Visual Servoing

Distributed: Tuesday, 2/22/2011, 3pm
Checkpoint: Friday, 2/25/2011, 3pm
Wiki Materials Due, and Briefings: Wednesday, 3/02/2011, 3pm

Objectives

In lecture you've heard about sensing generally, and about camera sensors specifically. You were also introduced to the Carmen package for planning and robot control. (Carmen is an acronym for the popular and widely-used "Carnegie Mellon Robot Navigation Toolkit.") Carmen allows you to abstract away most of the implementation-dependent details inherent in realizing a robot system that incorporates sensing, planning, navigation and control, freeing you to focus on higher-level issues when crafting algorithms that reason about your robot's observations of its environment and internal state.

Your objectives in this lab are to:

- Become familiar with the structure and use of the Carmen robot control package;
- Implement on-line digital image acquisition, along with an initial set of low-level image processing operators including blob detection and blob size and centroid estimation;
- Use the features extracted by your operators to aid in deciding how to control the robot, in this case to perform "visual servoing," by following a colored ball;
- Gain experience assessing your system's operating assumptions and robustness. You will write up the lab along with a discussion of your design assumptions and the conditions under which your implementation will likely fail.

Materials

For this lab you should have:

- One LogiTech QuickCam (spec. sheet on the RSS site), ball-joint mount and mount plate
- Two 1/4-20 screws and sliders
- *Guide to the Use of the Carmen Mobile Robot Control Package Within RSS*

Time Accounting and Self-Assessment:

Make a dated entry called "Start of Visual Servo Lab" on your Wiki's Self-Assessment page. Before doing any of the lab parts below, assign a number to describe your proficiency **as of the start of the lab**: 1=Not at all proficient; 2=slightly proficient; 3=reasonably proficient; 4=very proficient; 5=expert

- **Java**: How proficient are you at programming in Java?
- **Carmen**: How proficient are you at using Carmen?
- **Image Features and Visual Servoing**: How proficient are you at extracting low-level features from digital images, and using these features for visual servoing?

- **Assumptions and Failure Modes**: How proficient are you at characterizing the assumptions inherent in a system that you have designed, and clearly stating the likely failure modes of the system?

1 The Carmen Robot Control Package

Carmen is a distributed collection of modules (technically, processes, each running in a separate address space) that together implement autonomous robot control, and support user operations such as remote modification and monitoring of robot internal state. Carmen modules communicate through a networked, anonymous "publish/subscribe" mechanism, in which the only supported communication mechanisms are to publish data of interest to other subscriber modules, and/or subscribe to receive notification of data published by other modules. See Handout 4-B for a more comprehensive overview of Carmen.

A robot control system built on top of the Carmen toolkit will ordinarily invoke at least three processes to support autonomous operation:

- The MessageDaemon process (`message_daemon`) provides inter-process connectivity to all Carmen modules, ensuring that modules may publish any data, and that modules may subscribe to any data type (and be notified whenever new values of that data type become available).
- The ParamDaemon process (`param_daemon`) provides a central repository for robot configuration parameters (e.g., odometry scale factors and biases, maximum velocity constraints, controller gains, etc.). ParamDaemon reads the `carmen.ini` file at startup and publishes its contents to any interested subscribers via MessageDaemon. ParamDaemon also supports changing (and re-publishing) some parameters during robot operation.
- The RobotCentral process (`robot_central`) integrates the published outputs of various sensor modules, performing time-stamping and odometry-stamping of each captured sensor measurement and publishing the stamped values. RobotCentral also provides local response (or "reflex") behaviors for current or imminent collisions, e.g., bump sensor activation or fast-closing sonar returns.

Because we use the μOrcBoard for motor and sensor management, and because in this lab we are developing a semi-autonomous (i.e., human-monitored) visual servoing capability, our robot system will invoke four additional Carmen modules:

- The OrcDaemon process (`uorc_daemon`) abstracts away the details of controlling the robot into high-level, body-relative velocity commands, and makes current odometry available to any subscribers. OrcDaemon will also pass bump-sensor and sonar state updates to RobotCentral in future labs.
- The QuickCamDaemon process (`quickcam_daemon`) abstracts away the details of managing the camera state, including any captured images. QuickCamDaemon publishes the most recently acquired camera image from the QuickCam as a 2D array of RGB pixels.
- The VisualServo process (`VisualServo`) analyzes the most recent captured image, and publishes robot velocity commands intended to size and position the detected visual target (if any) in the robot camera's field of view.
- The VisionGUI process (`VisionGUI`) provides a graphical user interface (GUI) with which one or more users can remotely command the robot, and observe and modify its internal state.

We anticipate that during development MessageDaemon, ParamDaemon, QuickCamDaemon, OrcDaemon, RobotCentral, and VisionGUI will be restarted rarely, whereas VisualServo will need to be restarted frequently as you add and debug your ball detector and visual servoing algorithms.

We have provided complete, fully functional `MessageDaemon`, `ParamDaemon`, `OrcDaemon`, `QuickCamDaemon`, `RobotCentral`, and `VisionGUI` modules, and an incomplete `VisualServo` module, along with interfaces for all seven modules. Your task, through the use of these interfaces and by completing `VisualServo`, is to enable your robot to visually servo to a colored ball. To complete this lab, **you should need to modify and recompile only** `BlobTracking.java`, `Histogram.java` and `VisualServo.java` in the `VisualServo` module.

2 Useful Linux Commands

2.1 What Processes are running?

```
ps -ef | grep -e <regular-expression>
```

The ps command lists all running processes. Its output is being sent as the input to grep which takes an input and filters for lines that contain the query-string.

2.2 How do I kill a process?

```
sudo killall -9 <process-name>
```

This command will force-kill the running process. Only use this if the program is being unresponsive. Process name can be replaced by process ID number (PID), which can be found via ps (the second column is PID).

2.3 How do I check the END of a file?

```
tail <logfilename>
```

This command will let you look at the 10 last lines of a file. Use this to look at a process's logfile, especially when you expect the logfiles to be very big

2.4 How do I learn more about specific linux commands?

```
man <command>
```

This command will bring up the command's manual page. It will list the options and expected arguments. Alternatively, many commands will give you this information if you run

```
<command> --help
```

3 Mount Laptop and LogiTech QuickCam on the Robot

1. Mount your group's laptop on the back of your robot with velcro, as shown on the exemplar. Connect your laptop to the μOrcBoard using the provided cable.
2. We have also provided the μOrcTest module (uorc_test) which you may find useful for low-level debugging. Run μOrcTest to check that your μOrcBoard hardware and peripherals (motors, encoders, etc.) are configured as expected by Carmen. Note that μOrcTest runs independently of other Carmen modules (i.e., it does not require MessageDaemon). For safety, place your robot on blocks before execution.
3. Use the provided aluminum mounting plate to attach the camera to your robot.

4 Obtain and Incorporate the New Source

We have placed the Carmen binaries and Carmen class library (Carmen.jar) in the scripts directory under RSS-I-pub. These files have been included in your path.

The Carmen java source (which is compiled into Carmen.jar) is located in RSS-I-pub/carmen/src. The Carmen javadocs can be found in RSS-I-pub/carmen/docs and can be viewed with a web browser. You should not need to compile the Carmen source, but you will want to familiarize yourself with the classes and interfaces.

1. On the Sun workstation, one person should copy the VisualServo Lab java files (*.java and build.xml) from the ~/RSS-I-pub/labs/VisualServo/src directory to your working copy of your group repository (RSS-I-group), in the usual manner:

```
cd ~/RSS-I-pub
svn up
cd ~/RSS-I-group/trunk/
svn export ~/RSS-I-pub/labs/VisualServo/
svn add VisualServo
svn commit VisualServo -m "added new source for VisualServo lab"
```

2. Verify that you can compile the VisualServo source on the Sun workstation by running `ant` in `RSS-I-group/trunkVisualServo`.
3. If you are using Eclipse, you will have to add this lab source as a new package in your RSS-I-group project and include Carmen.jar in your build path.

5 Configure the Laptop for the QuickCam

1. If you have not already done so, plug in the camera to the laptop's USB port. The QuickCam should be auto-detected, and the `pwc` module should load automatically (courtesy of `/etc/init.d/hotplug`).
2. Verify that the required modules are loaded by running `lsusb | grep Logitech` at the shell prompt. You should see something resembling the following output:

   ```
   Bus 001 Device 004: ID 046d:0804 Logitech, Inc.
   ```

   If you do not see the above module list, wait a few moments and try again. If this doesn't work, try unplugging and reinserting the camera plug. If you have any problems please ask a TA.
3. Please see **Appendix A** at the end of this lab handout to finish configuring the camera.
4. Open a window with five tabs, each running a shell. In each tab run one of the five Carmen binaries, each compiled from C source. You will not need to modify or rebuild any of these binaries for this lab (if you are interested in compiling the source, ask the staff). Within each shell, start one foreground Carmen process, respectively:

   ```
   message_daemon
   param_daemon
   uorc_daemon
   quickcam_daemon
   robot_central
   ```

   You must start `message_daemon` first, then `param_daemon`, but the remaining processes can be started in any order.
5. Now, verify that the QuickCam is working on the laptop by opening a terminal and typing `camera_view`. This should open a window displaying a live camera feed. Test your setup by waving your hand in front of the camera and watching the on-screen video output.
6. Press `Control-c` to close each of the processes. Or you can use the script we've provided, `carmen-kill.sh`.

As an alternative to Step 4, you may use the script we've provided, `carmen-start.sh <camera?>`. If ANY argument is given, it will start all 5 processes. If there are no arguments, `quickcam_daemon` is not started, but the others are. All of these processes will run in the background; their logfiles can be found in `/var/tmp/carmen-logs`.

6 Running Code Remotely

1. You can also run this code remotely using your Sun workstation, given that your group's laptop is connected wirelessly to the public network. First, determine the IP address of your laptop by running `ifconfig ra0` (the IP of `ra0` is for the wireless). On your Sun workstation, open a terminal and SSH into your group's laptop:

```
ssh rss-student@<IP address of ra0>
```

2. Start the five Carmen processes as above. (If you want to start them individually, then you will need 5 separate terminal tabs each ssh-ing and starting a separate carmen process. We strongly recommend that you instead use the provided script carmen-start.sh <camera?>, which will start all of the processes in the background).
3. You can also run the camera_view process on the Sun workstation. In a new shell on your Sun machine, set the CENTRALHOST environment variable, which is used by Carmen to store the hostname of the machine that is running robot_central – in this case, your laptop:

   ```
   export CENTRALHOST=<IP address of ra0>
   ```

   Now run camera_view. You should see the live camera view. Why is the refresh rate slower than you observed on your laptop?
4. Now on the Sun, start the remaining Carmen processes from your RSS-I-group/VisualServo/ directory.

   ```
   ant run-visual-servo
   ant run-vision-gui
   ```

   The VisionGUI should show you an iconic overhead view of the robot, and a reduced version of the camera image. Congratulations! You are now ready to start implementing your solution to this lab.

   Both of these processes are reading CENTRALHOST from your terminal's environment. If CENTRALHOST is defined, it will be sent in as a command-line argument to the java files, overriding any default you may have set in your java files. You can check to see if CENTRALHOST has been set by running:

   ```
   echo $CENTRALHOST
   ```

   Also, note that if you change CENTRALHOST, you must recompile the code in order to see the change.
5. **Note:** Although it is not required for this class, you should be able to run code on the group laptop using your personal computer (as long as you are wirelessly connected to an RSS network), if you wish. Linux and Mac users can SSH into the group laptop from the command line; Windows users will need to download a SSH client such as SecureCRT or PuTTY. You will need to type in the full command:

   ```
   ssh <username>@<IP address of ra0>
   ```

   Viewing a GUI or camera feed remotely will require an X11 client such as X-Win and adding -X to the ssh command.

Deliverables: At the beginning of lab on Wednesday, a member of the staff will ask you to show that camera images are being received by the VisionGUI from VisualServo.

7 Use Carmen to Control the Robot

VisionGUI subscribes to and displays processed camera data (i.e., images) of class VisionImageMessage published by VisualServo. VisionGUI also draws an icon that represents the current robot state. You can also drive the robot around with VisionGUI; see VisionGUI.java for keybindings. Take the robot for a spin. Wave an object in front of the camera and see the most recent image change.

Deliverables: At the beginning of lab on Wednesday, a member of the staff will ask you to demonstrate driving your robot around using the VisionGUI running on your Sun machine.

8 Target Pixel Characterization

Your goal in this part of the lab is to write a method that detects, reasonably robustly, whether or not the colored target ball is being held in the camera's field of view. Read through this entire section before doing any coding or experiments.

In this section you will implement your blob segmentation algorithm in the BlobTracking class. After completing the segmentation algorithm you will add code to this class that computes the desired translational and rotational velocities for the robot to track the blob. This will be used by `VisualServo.java`. The method

```
public void apply(Image src, Image dest)
```

takes an input image from the Carmen camera handler, and returns a modified image to be displayed in VisionGUI.

You will confront significant image *noise* in this lab. Think about how you might combat noise through manipulation of your source image data, i.e., by some operation on each image as soon as it is received from the camera.

- RGB Value Stability
  1. Write code within `BlobTracking` to print the RGB pixel values reported by your camera when it is, and is not, imaging your target ball. (You may want to print only a few values near the center of the image for each video frame your camera captures, and/or print an average value over some central region.) *Record on the Wiki the sensed RGB values for the ball under a variety of lighting conditions:* near the hangar windows in bright and cloudy daylight; away from the windows; with the hangar lights on, and the ball fully illuminated; and with the ball in a shadow (for example, due to a large occluder placed between the ball and any significant light sources).
  2. We have provided a template (`Histogram.java`) for producing a histogram of the R, G and B values of an image. If the frame is W pixels wide, the histogram method creates a 2-D array with W buckets and three entries per bucket. These histograms tabulate, for each (discretized) value, the number of pixels with that value. This will help you gain intuition into which pixel representation you should use, and how you should design your BlobPixel classifier so that it is reasonably robust.

     A realtime histogram can be created by applying the method `Histogram.getHistogram()` to all of the images you display. Your job is to complete the `makeHistogram()` method which tallies the RGB values in the image. Do this, then enjoy your "real-time" histogram display in the VisionGUI window!

     The displayed histogram is fairly primitive; it just stacks the channels on top of each other. For extra credit you can improve the displayed histogram. For instance you can make it photoshop-like and blend the channels together. Or you can write the bins with the max values out to the display image (for easy calibration). You will need to modify `overlayHistogram()` to do this.
  3. For a frame in which the ball **is not present**, examine the histograms (one each for the R, G and B values) of all pixels in that frame. *Use* `printscreen` *to capture the VisionGUI window; post your screenshot to the wiki.*
  4. For a frame in which the ball **is present**, and occupies between a third and half of the frame area, *capture and examine the same set of histograms as above.*
  5. *Discuss the following on the Wiki:*

     How stable are the sensed RGB values as lighting conditions vary? As the camera-ball distance changes? At the apparent center of the ball versus near its limb? At the image center versus near its edge? What average RGB value and tolerance would include all of these sensed values as "positive" detection events? Would a detector using this average and tolerance be sufficiently discriminative to enable successful ball-following behavior in the hangar? Why or why not?

- HSB Color Space
  1. The `Image.Pixel` class has the ability to convert RGB pixel values to the HSB (Hue, Saturation, Brightness) color space, which represents color stimuli by the hue or pure color of the stimulus, the saturation or amount of black (white) in the stimulus, and the brightness or (roughly) equivalent monochrome intensity of the stimulus. Note that the hue value wraps from 0.0 to 1.0. Adapt `makeHistogram` such that it pays attention to the `hsbHistogram` variable. If it is true the histogram should be computed in terms of HSB, if it is false it should compute the RGB histogram. Now repeat your experiments above, characterizing the range of HSB values at pixels imaging your target ball under the same variety of lighting conditions, standoff distances, and on-axis/off-axis viewing conditions.

2. For a frame in which the ball **is not** present, *capture and examine the histograms of H, S and B values in that frame.*
3. For a frame in which the ball **is** present, and occupies between a third and half of the frame, *capture and examine the same set of histograms as above.*
4. *Discuss the following on the Wiki:* How stable are the sensed HSB values? On which of the HSB values would you base an improved ball detector? What value and tolerance would you use to achieve a reasonably high true-positive rate without unreasonably large numbers of false-positives?

- BlobPixel Method
  1. Using the detection criteria you arrived at above, write a method in `BlobTracking` that classifies each pixel in the image as a target blob (ball) pixel or not. Write code to modify the contents of the output image message so that blob pixels are converted to a pure, saturated color (e.g., for a red ball, Red=255, Green=0, Blue=0; for a yellow ball Red=255, Green=255, Blue=0), and non-blob pixels are converted to grey-scale pixels with luminance equal to the average of their RGB values (e.g., Red=Green=Blue=(R+G+B)/3). Be careful not to overwrite the RGB values as you compute the new ones! This technique preserves the general appearance of the scene while highlighting the detected element.
  2. We have included a Gaussian blur filter in `GaussianBlur.java`, which you may choose to utilize for preprocessing. Additionally, you can use the `param_edit` module to modify various system parameters (thresholds, etc.) at runtime. An example of using the Carmen `Param` class can be found in VisualServo.java.

- BlobPresent Method
  1. Now write a method in `BlobTracking` that uses your pixel-level classification to determine whether the ball is present in the overall image. (You can use simple pixel counting as the basis for your decision; we will see more sophisticated methods later in RSS, and you are welcome to experiment with such methods in this lab if you wish. We have included `ConnectedComponents.java` if you wish to try segmenting with connected components).
  2. Compute the ball area, in pixels, as well as the ball centroid, in pixel coordinates. Next, modify the output image message to highlight these pixels, for example by drawing their bounding box or using any other method you wish. Finally write code to indicate the ball centroid with an 8×8 pixel cross in some complementary color with vertical post and horizontal arms.

Deliverables: Your Wiki materials should include brief responses to the questions raised above. Also include screen shots created as you worked through the items above, with captions, specifically:

- *Four histograms (RGB, ball present and not present; HSB, ball present and not present).*
- *The VisionGUI window when no ball is present, showing the unmodified camera image.*
- *The VisionGUI window when the ball is present, showing the camera image with the highlighted ball pixels and indicated centroid.*

9 Visual Servoing

You are now poised to implement your robot's ball-following behavior.

- Calibrate Your Camera

  Write the output of the centroid and area of the detected ball to `System.out` or to a file. Now hold your target ball at a variety of known distances and headings from the robot origin (take into account the difference between robot and camera origins). For each image you know the ball's actual size, and a "Fix" (range and bearing) on the ball with respect to the camera. Use this information to derive a function `blobFix` that, given the output of `blobPresent`, reports the estimated range (in meters) and bearing (in signed radians, with zero signifying straight ahead) to the ball target.

- Visual Servo to the Ball

  Your robot should attempt to maintain a pose that faces the ball target and is 0.5m away (i.e., heading = 0 radians, range = 0.5m). You may use your code or the staff solutions from previous labs. If you choose to write your own controller, you may choose to structure it either as a velocity controller (i.e., command speed and direction) or as a pose controller (i.e., command body-relative position and heading). Formulate two error terms, one for each of the range and heading errors. Write a controller that drives both errors to zero by commanding the robot with appropriate motion commands.

  Carmen has some methods that you may use if you wish. For example, `robot.setVelocity(double translationalVelocity, double rotationalVelocity)` sends direct velocity commands to the wheels; this can be useful for stopping the robot. Some teams in the past have used the `robot.moveAlongVector(double distance, double theta)` method, which attempts to move the robot to a given location. Note that this method does have some limitations, which you will have to work around if you choose to use it. Because it doesn't block, Carmen allows you to call `moveAlongVector()` with a different command before the first one has completed. Furthermore, the method does not notify you when a motion is complete; you must determine this though other means, such as odometry.

Deliverables: Briefly describe on the wiki the performance of your visual servo algorithm. What is its characteristic response time? Does it oscillate? Is it susceptible to noise? Does it have reasonably low false negative (i.e., missed target) and false positive (i.e., hallucinated target) detection rates?

10 Run Fully Autonomously

Until now, you have been using a develop and debug cycle that includes a VisionGUI process, allowing you to see the image frames captured by the robot, the outputs of its pixel classifiers and ball detectors, etc. Also for your convenience (fewer code transfers to the laptop) we arranged by default for the VisualServo process to run on the Sun (though under Carmen, most processes can run almost anywhere as discussed above).

For this final part of the Lab, we ask you A) not to run VisionGUI, and B) to run VisualServo on your laptop rather than on the Sun. Now hold your ball target up in the robot's field of view. Everything should work just as before, probably with a lower-latency control loop. Congratulations! You have realized a fully autonomous, high-level, responsive behavior on your robot.

Deliverables: Post a description of your robot's operation. Does anything about its behavior differ from the previous run? Include in your briefing a demo video of your robot running fully autonomously. It should show a human moving the ball in front of the robot, and the robot servoing toward and away from the ball.

11 Assumptions and Failure Modes

One critical part of any engineering endeavor is a clear statement of your design assumptions, operating assumptions, and failure modes:

- The assumptions that you made in designing your solution, i.e. any assumptions you made about the functional components (including sensors) within your system.
- Your assumptions about the operating conditions under which those components will function as designed and as desired.
- The failure modes for your artifact, in other words, a clear statement of the conditions under which your system will not produce useful behavior.

Making an explicit statement about assumptions and operating conditions is useful for several reasons. First, it helps delineate which aspects of some overlying problem have been solved by the current effort, which is useful both for you and any others who wish to continue to improve the system's performance. Second, it "keeps you honest" in

the sense that an explicit statement of failure modes is a clear acknowledgement that one does not believe that one's system has solved every problem in the world. You might be surprised at the high degree to which such statements of acknowledgement are appreciated by your peers, for example, at technical conferences. Third, just as a general matter of scientific and engineering practice, it is difficult to make well-defined progress in a given problem area unless the people working in that area agree upon, and apply, qualitative and quantitative assessment metrics for their efforts.

In light of these general admonitions to good engineering practice, let us now turn to your specific efforts for this lab. In writing code for each of the lab parts you made certain operational assumptions about how the robot works, both at the hardware and software level. *In your Wiki materials, discuss the ways in which your operational assumptions can be made to fail in practice. What assumptions underlie your hardware and software? What assumptions does your software, architecture or implementation make about how the hardware works? Under what conditions will your system function as desired? Under what conditions is it likely to fail? For each assumption or failure mode, briefly discuss how you would generalize your implementation to either remove the assumption or to decrease the likelihood of the failure mode's occurrence.*

Time Accounting and Self-Assessment

After preparing your wiki materials, return to the Time and Assessment pages. Tally your total individual effort there, including time spent writing up your responses. Answer the "Self Assessment" questions again, post-Lab. Add the date and time of your post-lab responses.

Conclusion

The completed code and wiki materials (with linked images and videos) are due by **3pm on Wednesday, March 3**. Commit your software into your group's repository and link from your team's top-level page to your wiki materials for the lab.

As usual, be prepared to demonstrate your robot's behavior at the start of Lab on the due date.

This concludes the VisualServo Lab.

Appendix A: Changing Camera Settings

For acquiring images from the camera, we will be using `camunits` which was developed by Team MIT for the 2007 Darpa Urban Grand Challenge. If you are interested, more information about `camunits` can be found at http://code.google.com/p/camunits/. While you will not need to interact directly with the camunits code, you will be using one of the associated tools, called `camview` for camera adjustment and calibration.

To start `camview`, type the following in a terminal window:

```
camview
```

You should see a window that looks like Figure 1. Here, you can set up the camera to use custom settings.

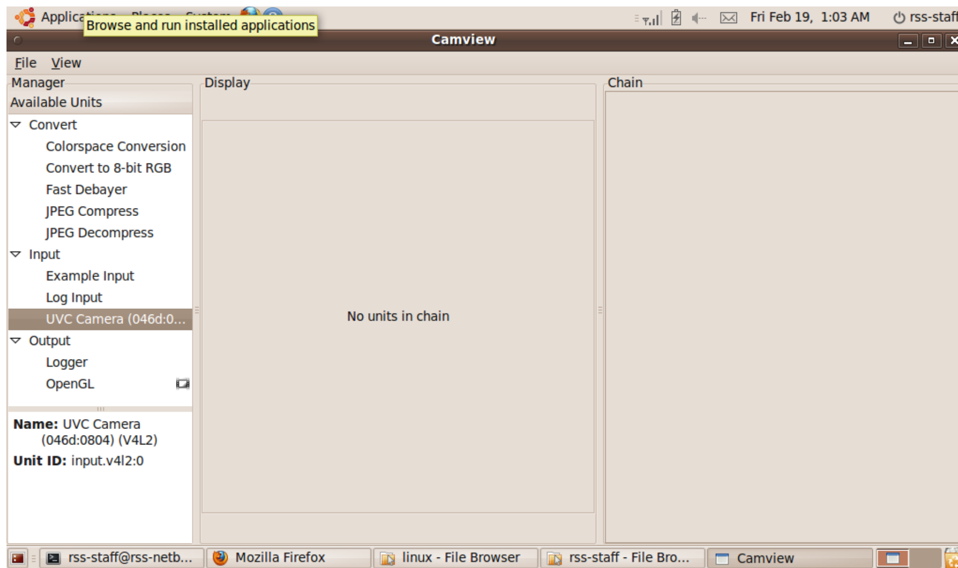

Figure 1: Camview dialog window

If the camera is connected correctly, you should see “UVC Camera (046d:0804)” as an available option under “Input”. Double click on this item to add it to the chain. If the camera is working, you should see the word “Streaming” in the box that appears and the Chain will start displaying a frame rate. If “Off” appears, check to make sure that the camera is connected and that no other process is using the camera (quit the `quickcam_daemon` process). If it looks correct, double click on “Convert to 8-bit RGB” then “OpenGL”. You should see something like Figure 2. Don’t freak out if the image is messed up. It’ll be okay.

Next, expand the UVC Camera box in the chain by clicking the triangle by “UVC Camera”. Under ”Format”, select“160x120 Motion-JPEG”. You should now see a (semi-normal) picture in the display. We’re asking that you choose to use a 160x120 resolution for now because the current processing code can handle this at about 3.5 frames per second. If you wish to write faster processing code that can handle a higher resolution, feel free to increase this later.

Feel free to play around with settings to find what works best for you. However, we have found the following to be useful:

- Uncheck the box for “White Balance Temperature, Auto”
- Set “Backlight Compensation” to 0

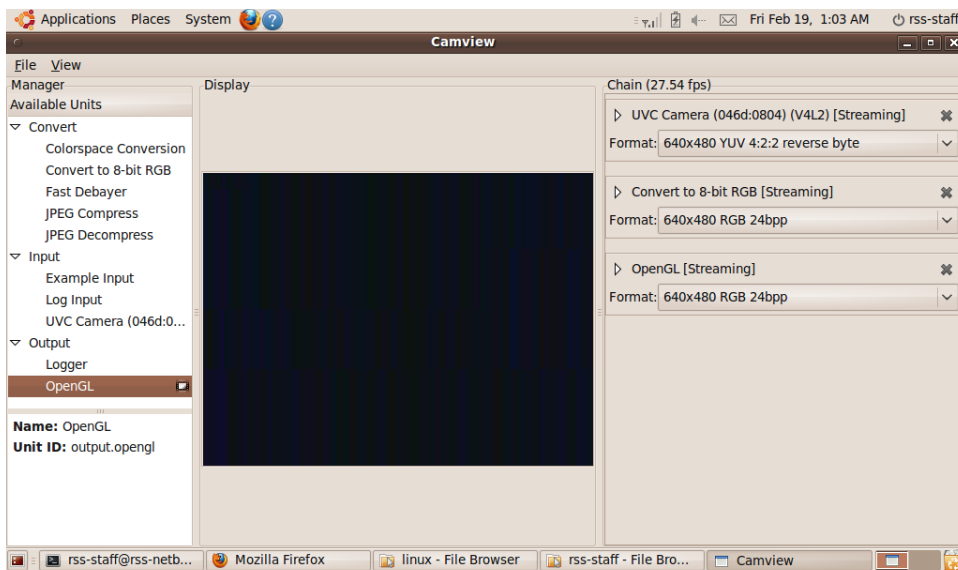

Figure 2: Camview dialog window with 3 units in a “chain”

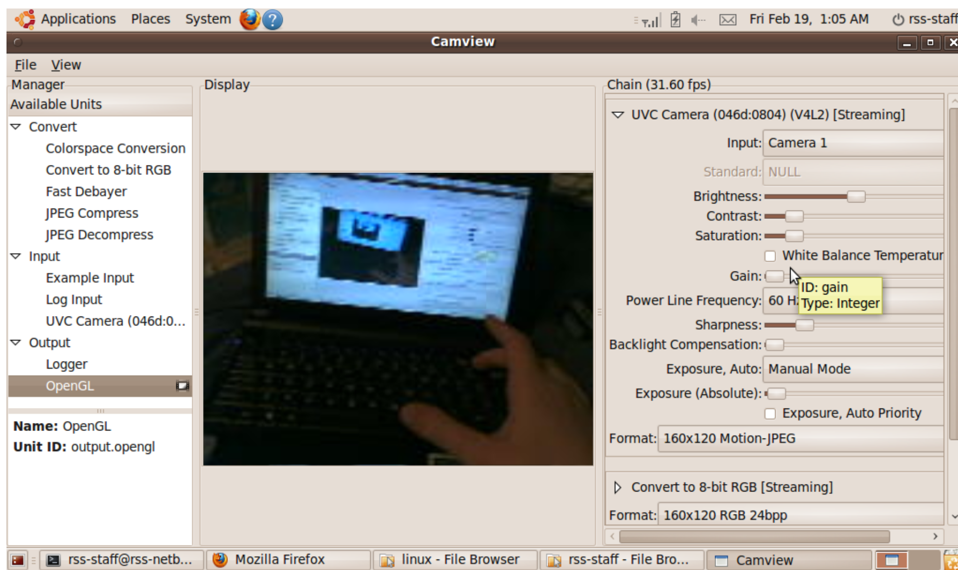

Figure 3: Changing the camera parameters

- Set “Exposure, Auto” to “Manual”
- Uncheck “Exposure, Auto Priority”
- Move the “Exposure, Absolute” Slider until the image looks good. Move the slider even if you think it already looks good.

Next, close the “OpenGL” unit. You may need to expand the right section of the screen to click the close box. You should now have 2 units in your chain. Save your chain as `camera_settings.xml` in `~/RSS-I-group/trunk/VisualServo`. Once you have saved your chain, close camunits.

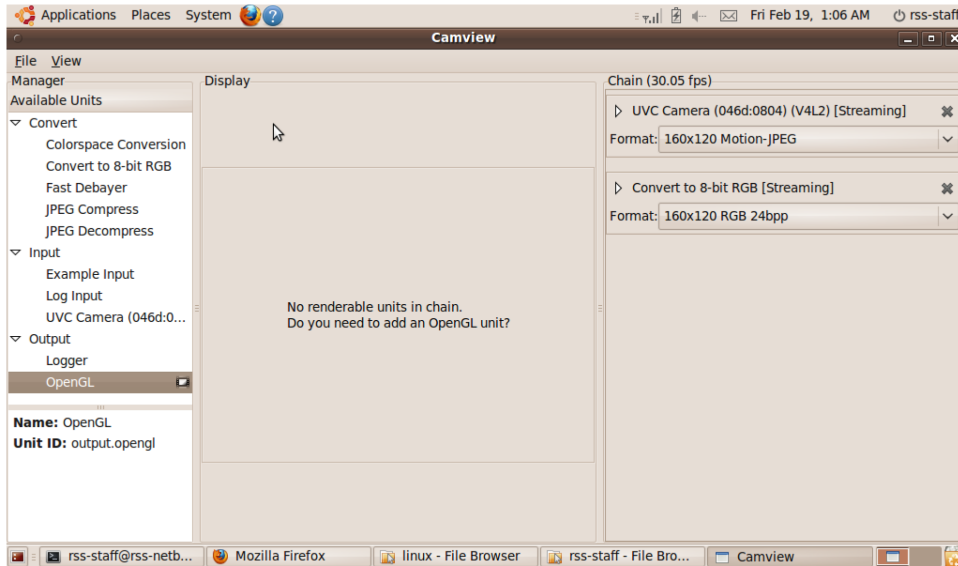

Figure 4: Saving a camunit chain

When you run `quickcam_daemon`, you will now need to run it from the directory where you saved your camera settings file. The daemon process will print a warning if it cannot find the file but will still load with default settings (which will usually be pretty bad). We

Note: Once you have set up your camera, you should be able to make multiple runs in the same environmental conditions. However, if the lighting changes, or if your code is not recognizing colors properly, or if you come in on a different day, you may need to reset the camera settings. To do this, you can open your settings file from within camunits, and follow the same steps from above.