Assignment 4

Gerritt Luoma

Problem 1

> Obtain a Logitech C270 camera (or equivalent) and verify that is detected by the Raspberry Pi USB driver.

> A: Show that it has been detected by using lsusb, lsmod and dmesg kernel driver configuration tool to make sure your Logitech C270 USB camera is plugged in and recognized. Do lsusb | grep -i logitech and prove to me (and more importantly yourself) with that output (screenshot) that your camera is recognized.

```
$ lsusb | grep -i logitech
// Confirmed my webcam make and model is correct
Bus 001 Device 003: ID 046d:0825 Logitech, Inc. Webcam C270
```

> B: Now, do lsmod | grep video and verify that the UVC driver is loaded as well (http://www.ideasonboard.org/uvc/).

```
$ lsmod | grep video
// Confirmed UVC video has been loaded
uvcvideo              110592  0
uvc                    12288  1 uvcvideo
videobuf2_vmalloc      12288  2 uvcvideo,bcm2835_v4l2
videobuf2_dma_contig    20480  3 bcm2835_codec,rpi_hevc_dec,bcm2835_isp
videobuf2_memops       12288  2 videobuf2_vmalloc,videobuf2_dma_contig
videobuf2_v4l2         32768  6
bcm2835_codec,rpi_hevc_dec,uvcvideo,bcm2835_v4l2,v4l2_mem2mem,bcm2835_isp
videodev              311296  7
bcm2835_codec,rpi_hevc_dec,videobuf2_v4l2,uvcvideo,bcm2835_v4l2,v4l2_mem2m
em,bcm2835_isp
videobuf2_common       73728  10
bcm2835_codec,videobuf2_vmalloc,rpi_hevc_dec,videobuf2_dma_contig,videobuf
2_v4l2,uvcvideo,bcm2835_v4l2,v4l2_mem2mem,videobuf2_memops,bcm2835_isp
mc                     61440  9
videodev,bcm2835_codec,snd_usb_audio,rpi_hevc_dec,videobuf2_v4l2,uvcvideo,
videobuf2_common,v4l2_mem2mem,bcm2835_isp
```

> C: To further verify, or debug if you don't see the UVC driver loaded in response to plugging in the USB camera, do dmesg | grep video or just dmesg and scroll through the log messages to see if your USB device was found.

```
$ dmesg
...
// Plugged in for the first time
[2353489.790568] usb 1-1.1: new high-speed USB device number 3 using
xhci_hcd
[2353490.091891] usb 1-1.1: New USB device found, idVendor=046d,
idProduct=0825, bcdDevice= 6.07
[2353490.091917] usb 1-1.1: New USB device strings: Mfr=0, Product=2,
SerialNumber=1
[2353490.091931] usb 1-1.1: Product: C270 HD WEBCAM
[2353490.091942] usb 1-1.1: SerialNumber: A1219F40
[2353490.200605] usb 1-1.1: Found UVC 1.00 device C270 HD WEBCAM
(046d:0825)
[2353490.325407] usbcore: registered new interface driver uvcvideo
[2353490.400438] usb 1-1.1: set resolution quirk: cval->res = 384
[2353490.400716] usbcore: registered new interface driver snd-usb-audio
// Disconnected the camera
[2353691.497092] usb 1-1.1: USB disconnect, device number 3
// Plugged back in
[2353699.120537] usb 1-1.1: new high-speed USB device number 4 using
xhci_hcd
[2353699.421908] usb 1-1.1: New USB device found, idVendor=046d,
idProduct=0825, bcdDevice= 6.07
[2353699.421935] usb 1-1.1: New USB device strings: Mfr=0, Product=2,
SerialNumber=1
[2353699.421949] usb 1-1.1: Product: C270 HD WEBCAM
[2353699.421960] usb 1-1.1: SerialNumber: A1219F40
[2353699.423685] usb 1-1.1: Found UVC 1.00 device C270 HD WEBCAM
(046d:0825)
[2353699.725742] usb 1-1.1: set resolution quirk: cval->res = 384
```

> D: Capture all output and annotate what you see with descriptions to the best of your understanding.

All output has been captured in parts A-C. The webcam connection is confirmed, the UVC drivers have been loaded, and the system logs corroborate that the device is connected with the drivers loaded.

Problem 2

> If you do not have cheese or camorama, do “sudo apt-get install cheese” on your Raspberry Pi board [you may need to first do sudo apt-get update]. Camorama is supported on Ubuntu, but may not be on the R-Pi, so use cheese instead on the R-Pi. If for some reason cheese does not work for you, use an alternative application (e.g., install with “sudo apt-get install camorama” or one of these - https://www.makeuseof.com/best-cameraapps-linux/). This should not only install nice camera capture GUI tools, but also the V4L2 API (described well in this series of Linux articles - http://lwn.net/Articles/203924/). Provide a summary log of your installation

I am first starting with trying to install `cheese` by using `sudo apt-get install cheese`. This will be a fairly large installation because cheese requires many GUI packages and I originally installed the headless version of Raspberry Pi OS on my dev board. The install will be ~850MB which is very large compared to what I was originally expected. The installation took about 3 minutes. Since I am still running headless, I will

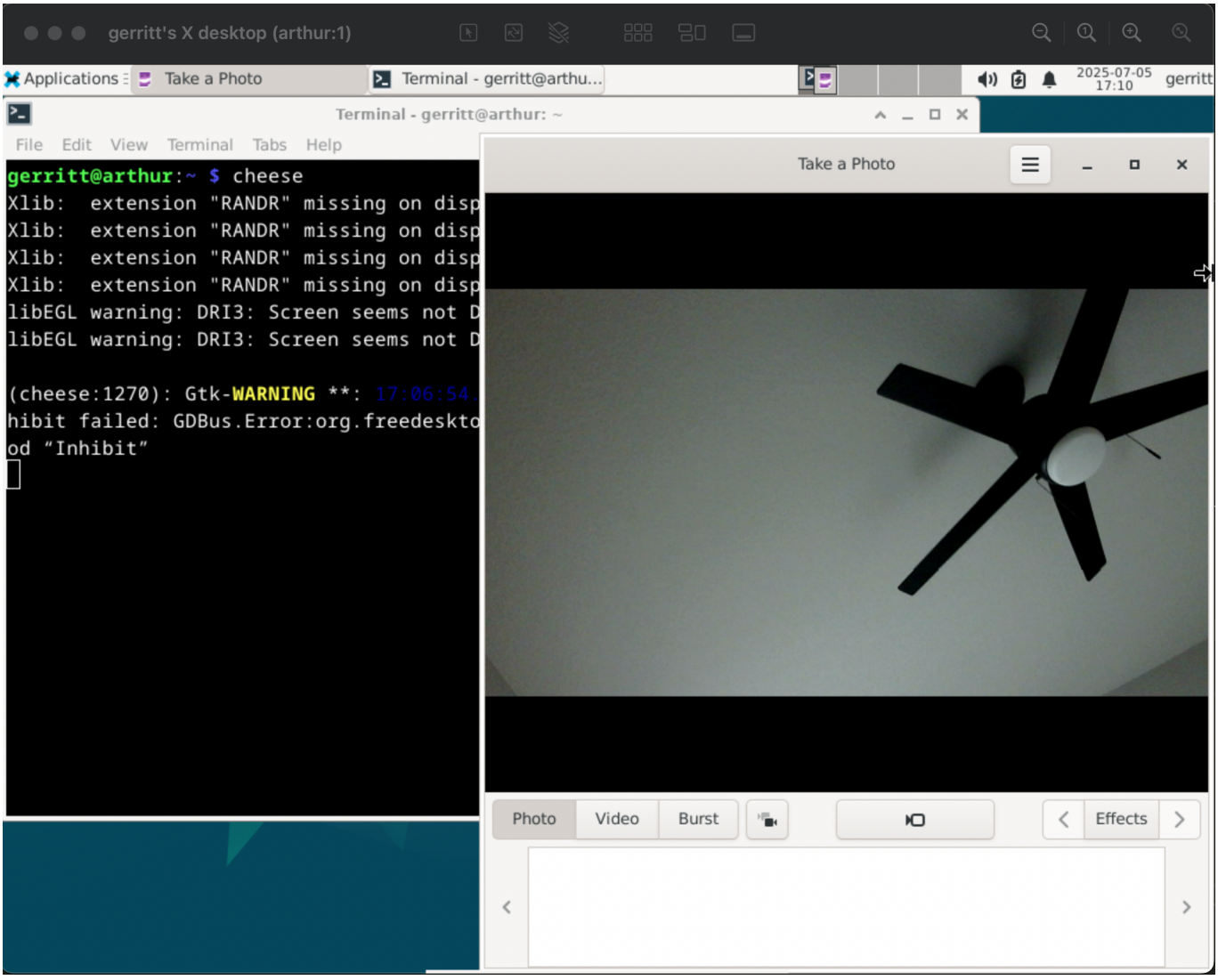
need to exit the current SSH session and restart the session with the `-X` flag to enable X-11 forwarding. I might need to enable some other settings within the pi as well.

After installing XQuartz on my laptop and trying some other workarounds, I am still unable to run cheese. I am getting segfault errors when cheese tries to start up. I am now going to try installing camorama with `sudo apt-get install camorama`.

`camorama` was unable to be installed via `apt` or `apt-get` potentially due to obselescence.

I moved over to using a vnc server so I can get a full desktop experience remotely. I ran `sudo apt install xfce4 xfce4-goodies tightvncserver` to install the dependencies. I then ran `echo "startxfce4" > ~/.xsession` and `chmod +x ~/.xsession`. To run the vnc server I needed to run `vncserver :1` on my SSH session and wait for it to return. Then from my macbook, I was able to open a finder window, hit `Cmd + K`, enter `vnc://<my raspberry pi ip>:5901`, and then log into the vnc session. I now have a graphical interface.

> A: Run camorama with no arguments and it should provide an interactive camera control session for your Logitech C2xx camera – provide a screen dump of the GUI.



> B: If you have issues connecting to your camera do a "man camorama" and specify your camera device file entry point (e.g. /dev/video0). Run camorama and play with Hue, Color, Brightness, White Balance and Contrast, take an example image, and take a screen shot of the tool and provide both in

> your report. If you need to resolve issues with your network, USB, or sudo and your account, research this and please do so.

Cheese appears to only have full screen effects like mirroring the image, distorting the image, etc. It doesn't have any hue or brightness alteration. I have resolved network, USB, and interface issues.

Problem 3

> Using your verified Logitech C270 camera on your Raspberry Pi, now verify that it can stream continuously.

> A: Using starter code, capture a raw image buffer for transformation and processing using example code such as simple-capture. This basic example interfaces to the UVC and USB kernel driver modules through the V4L2 API. Provide a screen shot to prove that you got continuous capture to work with V4L2. To display images you have captured, you will find “eom” to be quite useful, which can be installed with “sudo apt-get install eom”.

Below is the output from running the starter code in `simple-capture`:

```
$ sudo ./capture
FORCING FORMAT
allocated buffer 0
allocated buffer 1
allocated buffer 2
allocated buffer 3
allocated buffer 4
allocated buffer 5
frame 1: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 2: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 3: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 4: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 5: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 6: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 7: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 8: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 9: Dump YUYV converted to RGB size 153600
```

```
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 10: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 11: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 12: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 13: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 14: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 15: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 16: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 17: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 18: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 19: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 20: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 21: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 22: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 23: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 24: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 25: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 26: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 27: Dump YUYV converted to RGB size 153600
```

```
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 28: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 29: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
frame 30: Dump YUYV converted to RGB size 153600
wrote 230400 bytes
time_error.tv_sec=0, time_error.tv_nsec=0
```

When logging into my VNC session and using `eom` to look through the images, I am noticing that the first ~15 images taken by the camera are fully black and after that all of the remaining images are valid captures of my office. I am wondering if it is because the camera takes time to go from idle to capturing and the program is going too fast for the camera to keep up.

I tried adding a thread sleep after initializing the camera/starting the capture and I am still not seeing an improvement. There might be a few less black images but there are still a few that are completely devoid of data at the start of the capture.

> B: Download starter code (simple-capture-1800) that captures 1800 frames from your camera using V4L2 and run it as is first to test the code (use “make clean”, “make”, and run – note that make clean will remove test frames). Modify the code so that it uses syslog rather than printf to the console for tracing. Then, compare frame rate for PGM (graymap) and PPM (color) image write-back to the frame sub-directory on your flash filesystem and provide analysis of average and worst-case (lowest) frame rate seen for each.

After running the code, I am seeing that the program differs slightly from the expected behavior as laid out by the question. This program by default only captures 182 frames as opposed to 1800 frames outlined in the problem statement. It has the same output as the `simple-capture` program as well. The program will take a total of 189 captures but the first 7 are always discarded based off the code. I am wondering if this is because of the issue that I noted in part A of this problem. The images are captured at a frequency of 1Hz.

After updating the program to use `syslog` instead of `printf`, I also added in timing functionality to the `read_frame()` function. This function handles **reading** the next frame from the camera, processing it, and writing it to a file which is the complete cycle taken for each frame. I have captured the logs running with `COLOR_CONVERT_RBG` both defined and undefined and captured them to `ppm-output.log` and `pgm-output.log` respectively.

I have written a python script, `analyze_logs.py`, and have placed it in the `simple-capture-1800` directory. First, to analyze the default behavior of the program with it outputting the images to grayscale, you can run `python3 analyze_logs.py` and when prompted, input `pgm-output.log`. This will produce the following output:

```
--- Analysis Results for: pgm-output.log ---
Mean read_frame time: 0.008402 seconds
Min read_frame time: 0.007256 seconds
Median read_frame time: 0.008393 seconds
```

```
Max read_frame time: 0.011143 seconds
----------------------------------------
Mean read_frame FPS: 119.021305 fps
Min read_frame FPS: 137.816979 fps
Median read_frame fps: 119.139811 fps
Max read_frame fps: 89.742439 fps
----------------------------------------
```

Ass seen here, the average time for reading, altering, and storing an image is 0.0084 seconds which translates to ~119.02 frames per second. Meanwhile, the worst case read time for a frame was 0.0111 seconds which translates to ~89.74 frames per second which is nearly a 30 frame per second drop from the average. In a soft real time environment this might be acceptable but for hard real time if the period was 0.01, the deadline would have been missed causing potentially catastrophic results.

Next, when analyzing `ppm-output.log` in the same way, the results produced the following:

```
--- Analysis Results for: ppm-output.log ---
Mean read_frame time: 0.034039 seconds
Min read_frame time: 0.029290 seconds
Median read_frame time: 0.034055 seconds
Max read_frame time: 0.036320 seconds
----------------------------------------
Mean read_frame FPS: 29.378185 fps
Min read_frame FPS: 34.141345 fps
Median read_frame fps: 29.364264 fps
Max read_frame fps: 27.533040 fps
----------------------------------------
```

It can be seen that when converting to RGB it takes much longer than just going to gray scale. This is most likely due to the amount of data being written to disk and the additional processing needed for each pixel. The average frame time was 0.03404 seconds, or 29.378 frames per second, with a worst case frame time of 0.3632 seconds, or 27.53 frames per second. It seems like a T of 0.05 seconds, or 20 frames per second, could be a good candidate.

Problem 4

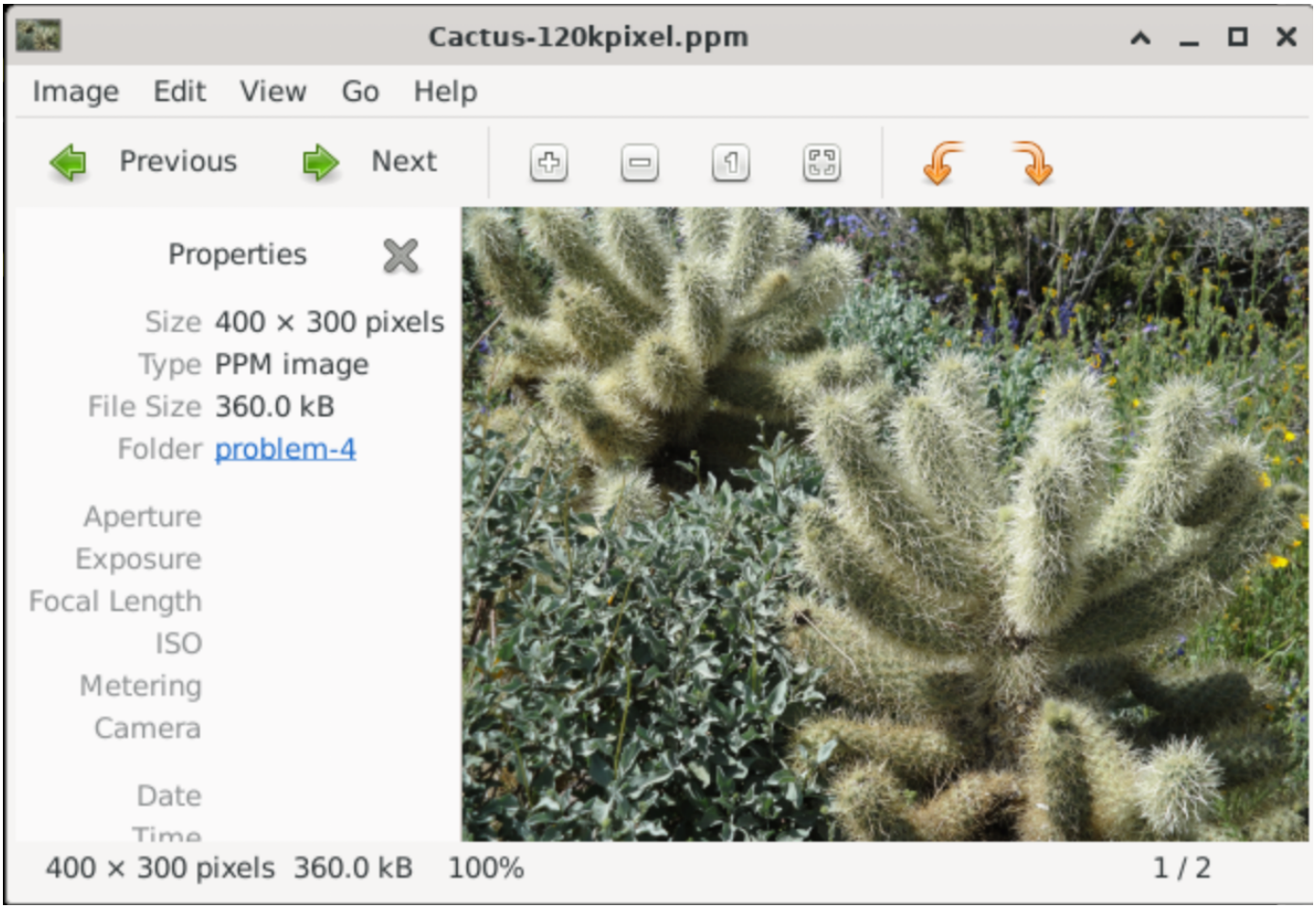
> Choose ONE continuous transformation for each acquired frame such as color conversion (Lecture-Cont-Media.pdf, slide 5), conversion to grayscale (slide 6), negative image (new_pixel = saturation – pixel), brightness/contrast adjustment (e.g., /cbrighten/brighten.c), or sharpening (/sharpen-psf/sharpen.c) and consider how to apply this to a real-time stream of image frames acquired from your camera rather than just one image file (PPM in the examples).

I have chosen to use the provided `sharpen` example to test sharpening images as a part of real time processing.

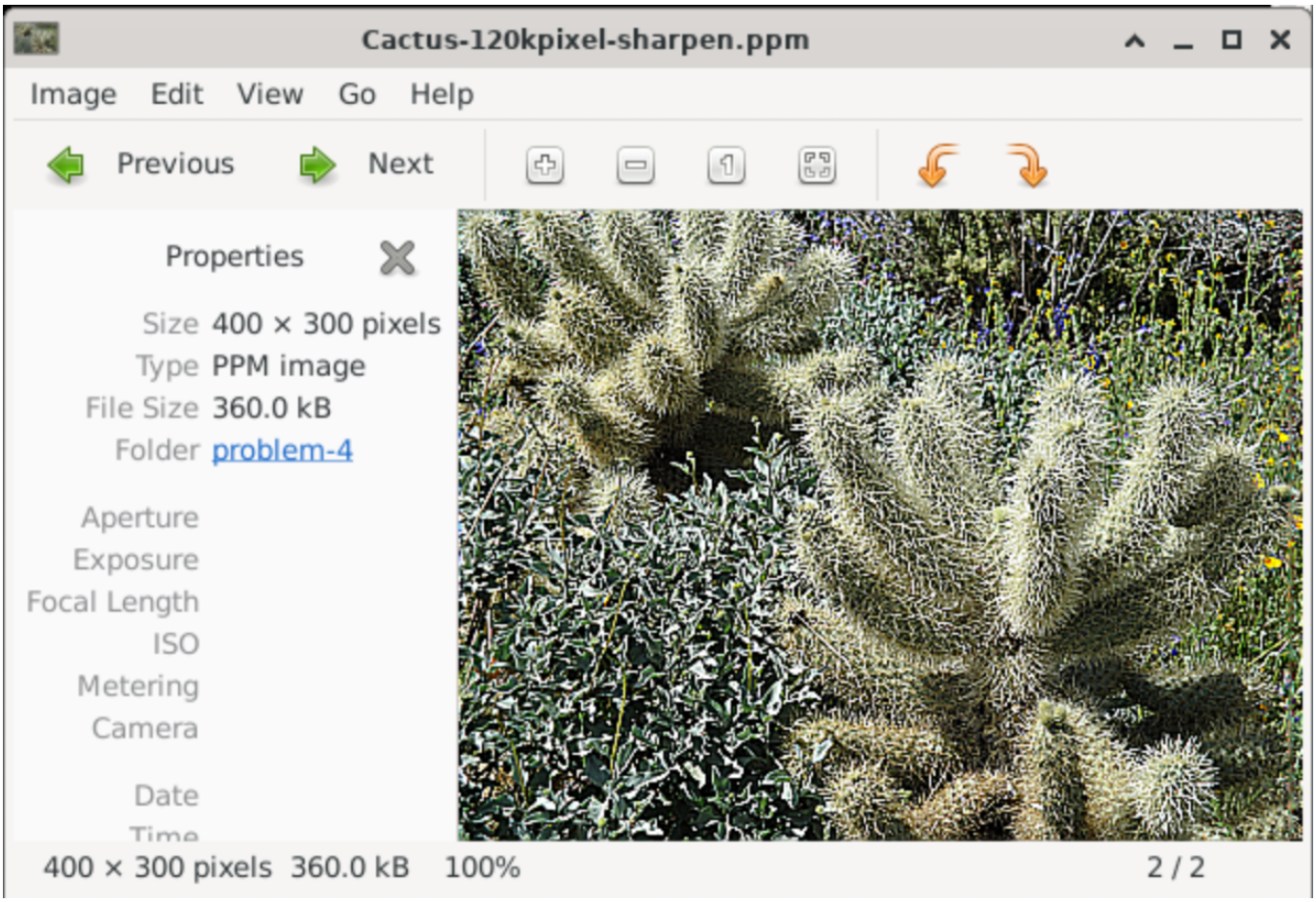
> A: Show a screen shot to prove you built and ran the code and a before and after transform example image.

I have updated the makefile so that it will build the `sharpen.c` file. This file takes in two inputs. One to provide the path to an input image which has to be `400x300` exactly and in the .ppm format and one to provide the path to where the resulting transformed image should be output which must also be a .ppm format.

I will be using the provided starter image, `Cactus-120kpixel.ppm`, to test sharpening an image and timing it. The starting image is:



Then, after building and running `sudo ./sharpen Cactus-120kpixel.ppm Cactus-120kpixel-sharpened.ppm` I get the resulting image:



The original image appears to have been a little blurry and after the transformation was performed, the image now appears with more distinct edges on the edges of objects in the image. It also appears a bit more distinct when transitioning from object to object making like colors more similar and different colors more distinct.

> B: Provide a detailed explanation of your code and research uses for the continuous transformations and describe how this might be used in a machine vision application.

The way that this code works is as follows:

1. The initial `.ppm` image and the output `.ppm` image are both opened with the input image being opened with the Read Only flag and the output image being opened with the Read/Write and Create flags meaning a new file will be created when the program is ran.
2. The header of the input image is parsed which is 21 bytes long.
3. The input image is then loaded into three individaul buffers, `R[], G[], B[]` which are the 3 color channels. Each of these buffers is 120,000 bytes long since the expected input image is 400x300. The color channels are also loaded into `convR[], convG[], convB[]` which are the same size.
4. The program then loops through each pixel in the image other than the first and last pixel of each row and column and convolves the each RGB value of the pixel with the RGB value of all of the surrounding pixels multiplied by a Point Spread Function.
5. After looping through each of the pixels and storing the convolved values in `convR[], convB[], convB[]`, it writes the original header into the output file and then outputs the RGB sets back into the image.

Potential uses for an image sharpening algorithm in a machine vision application:

1. Sharpening an image enhances the distinct features within an image. This makes it easier for other algorithms to distinguish objects from the background or other objects. This will also improve the accuracy of object detection if it is easier.
2. A sharpened image is easier to extract features of an image. This will improve both training and run time performance of tasks like image classification or segmentation.
3. Sharpening will also help with reducing blur. If the camera is blurry for any reason, sharpening can help un-blur the image a bit to improve the image quality.

> C: What was the best frame rate for continuous transform processing for the transform you chose to investigate?

I have added a new file named `sharpen-240.c` that performs the same operations as `sharpen.c` but instead it now will load the file, process it, and store the file a total of 240 times. During this time, the program will write to `syslog` the following message each iteration: `Processing time: XXXX seconds`. I have added in two new define settings, `LOAD_ONCE` and `WRITE_IMAGE`, to allow for different levels of testing. When you only have `WRITE_IMAGE` defined, the program will load the file, process it, and write it back each iteration which is expected to take the most amount of time. The second permuation is having neither defined which will cause the program to read the file and process it each iteration but it will not write the file back. This will take a medium amount of time per iteration. The final permutation that I will test is having only `LOAD_ONCE` defined which will cause the program to load in the image from the source .ppm file once before entering the processing iterations which will cause it to only process the image each iteration which should cause it to be relatively fast. I have added in a python processing file called `analyze_logs.py` which is ran in the same way as the python file in problem 3 and will generate the same output. The results can be found below:

The test with `WRITE_IMAGE` defined and `LOAD_ONCE` undefined can be found in the file `load-and-store-all.log`.:

```
$ python3 analyze_logs.py
Please enter the name of the log file to analyze (e.g., pgm-output.log):
load-and-store-all.log

--- Analysis Results for: load-and-store-all.log ---
Mean read_frame time: 0.771513 seconds
Min read_frame time: 0.759106 seconds
Median read_frame time: 0.771093 seconds
Max read_frame time: 0.812216 seconds
----------------------------------------
Mean read_frame FPS: 1.296154 fps
Min read_frame FPS: 1.317339 fps
Median read_frame fps: 1.296860 fps
Max read_frame fps: 1.231200 fps
----------------------------------------
```

The results for this test are fairly surprising. I believe it is because loading and storing the image through flash operations is extremely slow. This would be borderline unusable for most projects.

The test with neither defined can be found in the file `load-and-no-store.log`.:

```
$ python3 analyze_logs.py
Please enter the name of the log file to analyze (e.g., pgm-output.log):
load-and-no-store.log

--- Analysis Results for: load-and-no-store.log ---
Mean read_frame time: 0.321949 seconds
Min read_frame time: 0.314823 seconds
Median read_frame time: 0.322284 seconds
Max read_frame time: 0.396265 seconds
----------------------------------------
Mean read_frame FPS: 3.106085 fps
Min read_frame FPS: 3.176388 fps
Median read_frame fps: 3.102853 fps
Max read_frame fps: 2.523564 fps
----------------------------------------
```

When running the test without storing the file after processing, the average time is improved by over 50% but the worst case time is still around 2.52 frames per second which is not the best. I believe once again that the bottle neck is the flash operations. When reading from the camera it is much quicker which allows for the transformation to be quicker.

The test with `LOAD_ONCE` defined and `WRITE_IMAGE` undefined can be found in the file `load-once-store-none.log`.:

```
$ python3 analyze_logs.py
Please enter the name of the log file to analyze (e.g., pgm-output.log):
load-once-store-none.log

--- Analysis Results for: load-once-store-none.log ---
Mean read_frame time: 0.025209 seconds
Min read_frame time: 0.025128 seconds
Median read_frame time: 0.025198 seconds
Max read_frame time: 0.025671 seconds
----------------------------------------
Mean read_frame FPS: 39.667795 fps
Min read_frame FPS: 39.796243 fps
Median read_frame fps: 39.686477 fps
Max read_frame fps: 38.954462 fps
----------------------------------------
```

The results here are much better with the processing of the frame taking on average 0.25 seconds allowinf for a frame rate of almost 39.66 frames per second. This still is taking quite a bit of time. I am going to try adding the `-O3` optimization to improve the processing further. I will put it under the file name `load-once-store-none-O3-optimization.log`.

```
$ python3 analyze_logs.py
Please enter the name of the log file to analyze (e.g., pgm-output.log):
```

```
load-once-store-none-O3-optimization.log

--- Analysis Results for: load-once-store-none-O3-optimization.log ---
Mean read_frame time: 0.004438 seconds
Min read_frame time: 0.004418 seconds
Median read_frame time: 0.004434 seconds
Max read_frame time: 0.004858 seconds
----------------------------------------
Mean read_frame FPS: 225.347034 fps
Min read_frame FPS: 226.346763 fps
Median read_frame fps: 225.529995 fps
Max read_frame fps: 205.846027 fps
----------------------------------------
```

This is a much better performance simply by adding the `-O3` comiler flag. I am certain that if I were to apply the optimization to the other runs I would see a similar performance boost. This is a total of an 82.4% performance boost giving hundreds of frames per second which is more than adequate for running some form of machine vision.

> Note that your real bottleneck may be writing your frames to your flash file system device, so try disabling frame write-back to the frame sub-directory and transforming frames, but not saving them.

Problem 5

> Using a Logitech C270 (or equivalent), choose ONE real-time frame transformation and measure the frame processing time (WCET). I have made some videos on the "L-N9.#-Final-Project" (found in Lightboard-Studio-rough-cuts) that demonstrate the analysis of a single thread design (simple-capture-1800) and use of syslog for tracing (CW21-C4-simplecapture-1800-starter-code-for-project) and you may also find simple-capture1800-syslog/ useful.

> A: Please implement and compare transform performance in terms of average transform processing time and overall average frame rate for acquisition, transformation and writeback of only the transformed frame. Note the average for each step clearly (acquisition, processing, write-back).

I have rewritten the sharpening starter code to instead be ran within the provided `simple-capture-1800-syslog` code. For the sharpening to work, at least how it was provided, the image must first be converted from YUYV to RGB. Once the image is in RGB, it is sharpened using the provided algorithm. I had to update the algorithm to handle the RGB channels all being in the same contiguous array of data which for the most part was reducing the number of multiplications and updating the indexes for the convolution of pixels.

When the program runs, it sharpens each frame and stores them into the `frames/` directory under the name `postXXXX.ppm` with XXXX incrementing each frame.

I have added in a new `analyze_logs.py` file and have saved the system logs of a run of just saving the transformed image. When the run completed, the c program showed the following stats:

```
$ sudo ./capture
FORCING FORMAT
```

```
allocated buffer 0
allocated buffer 1
allocated buffer 2
allocated buffer 3
allocated buffer 4
allocated buffer 5
Running at 30 frames/sec
at 0.000000
at 0.000000
at 0.000000
at 0.000000
at 0.000000
at 0.000000
at 0.000000
at 82971.075102
at 82971.119398
Total capture time=90.540054, for 1802 frames, 19.891749 FPS
```

The C program showd that the average frames per second over the course of the program were 19.891749 frames per second. After running the system logs through my analyzing script, it showed the following:

```
$ python3 analyze_logs.py
Please enter the name of the log file to analyze: transform-writeback.log

--- Statistics for Image Acquisition Time ---
Mean: 0.000017 seconds
Min: 0.000008 seconds
Median: 0.000015 seconds
Max: 0.000095 seconds
----------------------------------

--- Statistics for Image Conversion Time ---
Mean: 0.008994 seconds
Min: 0.008300 seconds
Median: 0.008422 seconds
Max: 0.025906 seconds
----------------------------------

--- Statistics for Image Write Time ---
Mean: 0.007305 seconds
Min: 0.001534 seconds
Median: 0.001998 seconds
Max: 2.200303 seconds
----------------------------------

--- Statistics for Total Time Per Frame (Combined) ---
Mean: 0.016318 seconds
Min: 0.009946 seconds
Median: 0.010598 seconds
Max: 2.208707 seconds
----------------------------------
```

```
Estimated Frames Per Second (FPS): 61.28
```

These results show that, on average, the longest running part of the program is the conversion from YUYV to RGB to Sharpened RGB with an average time of 0.008994 seconds. The worst case time, however, comes from the image write back step of the processing loop with a worst case time of 2.208707 seconds. This most likely occurs when the flash needs to perform a sector erase before it can be written to causing the system to seize up. That being said, when averaging together the total processing time of each frame the estimated possible frames per second is 61.28 frames per second. This is much higher than the FPS reported by the program at runtime, 19.89 FPS, most likely because of two reasons:

1. The camera being used is only able to support a maximum frame rate of 30 FPS. Any faster and the camera will be the main limiting factor of the program.
2. The current method of timing for the program is using `nanosleep()` to always sleep enough time per iteration to cause the maximum frame rate of the program to be 30 FPS. The issue here is that it is not accounting for the processing time of the frames so if a frame took up half of that iteration's 30 FPS time window, the program will still perform the full sleep causing the timing of the thread to be much slower.

> B: Please implement and compare transform performance in terms of average frame rate for transformation and write-back of both the transformed frame and captured.

I have added in a new flag to the program, `DUMP_RGB`, that will cause the program to save both the regular unsharpened image and the processed sharpened image to the file system each iteration. The base RGB frames are saved under the names `testXXXX.ppm` in the `frames/` directory. The output from this test run was as follows:

```
$ sudo ./capture
FORCING FORMAT
allocated buffer 0
allocated buffer 1
allocated buffer 2
allocated buffer 3
allocated buffer 4
allocated buffer 5
Running at 30 frames/sec
at 0.000000
at 0.000000
at 0.000000
at 0.000000
at 0.000000
at 0.000000
at 0.000000
at 84067.081015
at 84067.128470
Total capture time=120.499455, for 1802 frames, 14.946126 FPS
```

As you can see, when writing both frames to memory the average frames per second takes a considerable hit dropping almost 5 frames per second. While the program was running, I was noticing that the syslog output was lagging consistently. I believe this was because of the fact the program was writing double the amount of files causing the flash sector erases to become more frequent. Below is the output of running the `analyze_logs.py` program on the `transform-and-original-write.log` output:

```
$ python3 analyze_logs.py
Please enter the name of the log file to analyze: transform-and-original-
writeback.log

--- Statistics for Image Acquisition Time ---
Mean: 0.000015 seconds
Min: 0.000008 seconds
Median: 0.000014 seconds
Max: 0.000079 seconds
----------------------------------

--- Statistics for Image Conversion Time ---
Mean: 0.008870 seconds
Min: 0.008339 seconds
Median: 0.008460 seconds
Max: 0.031969 seconds
----------------------------------

--- Statistics for Image Write Time ---
Mean: 0.023965 seconds
Min: 0.003164 seconds
Median: 0.003763 seconds
Max: 2.343479 seconds
----------------------------------

--- Statistics for Total Time Per Frame (Combined) ---
Mean: 0.032850 seconds
Min: 0.011628 seconds
Median: 0.012484 seconds
Max: 2.351889 seconds
----------------------------------

Estimated Frames Per Second (FPS): 30.44
```

The time taken for acquiring the image and converting the image remained roughly the same compared to the prevous run which was expected. The average time for writing the files to the filesystem have more than tripled, however, due to the flash sector erases. Because of this, the theoretical average frame rate for the program was 30.44 FPS but it would have had multiple missed deadlines throughout the course of the program.

> C: Based on average analysis for transform frame only write-back, pick a reasonable soft real-time deadline (e.g., if average frame rate is 10 Hz, choose a deadline of 100 milliseconds) and convert the processing to SCHED_FIFO and determine if you can meet deadlines with predictability. Note, here

> are some examples of monotonic service analysis and jitter (here). The drift is shown as a red polynomial trend line and jitter is a plot of raw delta-t data compared to expected.

I have created a new file in the `problem-5/` directory named `capture-srt.c`. This is a combination of the `capture.c` file in the same directory and my code from problem 5 of assignment three which was based off Prof. Siewert's `seqgen3.c` example code. I have turned the main thread into a `SCHED_FIFO` pthread and have changed the `mainloop()` function to wait on a semaphore for it to be released by a sequencer. I am then creating a sequencer that runs based off a timer firing at a rate of 20 Hz which will release the main thread each time the sequencer runs.

I am choosing a deadline of 50ms, or a frequency of 20Hz for the image processing thread. I am pickign this deadline because even though the code was running at ~19FPS in my previous test runs, I believe part of the lower FPS was due to the scheduling being done my a `nanosleep()` call that didn't take into account the time taken to process the image each iteration leading to the thread running slower than it potentially could have been. After running the program I received the following in standard output:

```
$ sudo ./capture-srt
FORCING FORMAT
allocated buffer 0
allocated buffer 1
allocated buffer 2
allocated buffer 3
allocated buffer 4
allocated buffer 5
at 0.000000
at 0.000000
at 0.000000
at 0.000000
at 0.000000
at 0.000000
at 0.000000
at 89573.814515
at 89573.851423
Total capture time=90.062120, for 1802 frames, 19.997308 FPS
```

This is showing that the thread is running almost exactly at 20 Hz which is what I programmed the sequencer to run at which seems to be promising. I also captured the system logs for the program and analyzed them using my `analyze_logs.py` program:

```
$ python3 analyze_logs.py
Please enter the name of the log file to analyze: soft-realtime.log

--- Statistics for Image Acquisition Time ---
Mean: 0.000016 seconds
Min: 0.000004 seconds
Median: 0.000015 seconds
Max: 0.000078 seconds
-----------------------------------
```

```
--- Statistics for Image Conversion Time ---
Mean: 0.008701 seconds
Min: 0.008278 seconds
Median: 0.008369 seconds
Max: 0.029224 seconds
-----------------------------------

--- Statistics for Image Write Time ---
Mean: 0.006352 seconds
Min: 0.001552 seconds
Median: 0.001889 seconds
Max: 1.510366 seconds
-----------------------------------

--- Statistics for Total Time Per Frame (Combined) ---
Mean: 0.015071 seconds
Min: 0.009942 seconds
Median: 0.010296 seconds
Max: 1.518755 seconds
-----------------------------------

Estimated Frames Per Second (FPS): 66.35
```

We can see that the average time spent processing the images was 0.015071 seconds which would come out to an average of 66.35 FPS if the thread wasn't gated by the semaphore. This average processing time is well below the deadline of 0.05 seconds to achieve a frame rate of 20 Hz. It should be noted that the maximum write time is still 1.5 seconds which means at least one deadline was missed over the course of the run. I have confirmed that all 1802 frames are present in the frames directory meaning all frames were captured and stored despite having at least one missing deadline.