Prerit Oberai
Ayush Dhoot

ECEN 5623
April 2, 2019

Exercise 4

Introduction

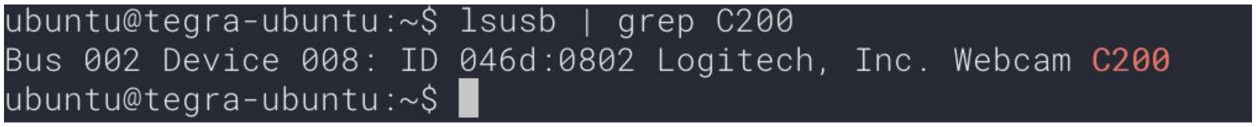
The purpose of this exercise is to gain an understanding of the use of cameras for computer vision applications for emergent real-time systems such as intelligent transportation, but also for traditional real-time instrumentation applications including machine vision, digital imaging for science and defense, and avionics instrumentation. We used a **Jetson TK1 board** for the code that we wrote in this exercise and all results obtained should be reproducible with the appropriate reference material and included scripts.

Problem 1

[10 points] Obtain a Logitech C200 camera or equivalent and verify that is detected by the DE1-SoC, Raspberry Pi or Jetson Board USB driver. For the Jetson, do **lsusb | grep C200** and prove to the TA (and more importantly yourself) with that output (screenshot) that your camera is recognized. For systems other than a Jetson, do **lsmod | grep video** and verify that the UVC driver is loaded as well (http://www.ideasonboard.org/uvc/). 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. Capture all output and annotate what you see with descriptions to the best of your understanding.

Solution:

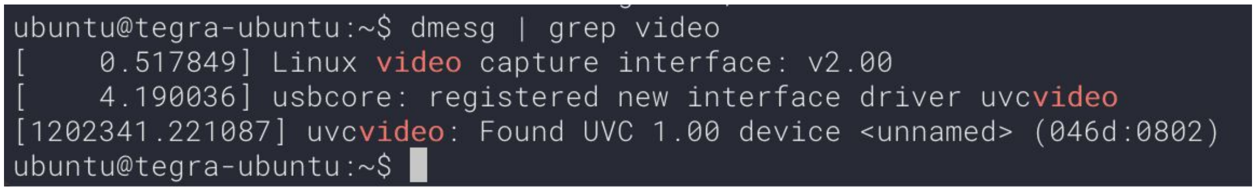
For this problem, we obtained a Logitech C200 camera from the TA's and plugged it into our Jetson TK1 board via the USB port. The following is a screenshot proving that it is recognized by the board and on Bus 002 as Device 008.



```
ubuntu@tegra-ubuntu:~$ lsusb | grep C200
Bus 002 Device 008: ID 046d:0802 Logitech, Inc. Webcam C200
ubuntu@tegra-ubuntu:~$
```

Figure 1: Proof of Logitech C200 on Ketson Board

The following verifies that the UVC driver is loaded in response to plugging in the USB camera.



```
ubuntu@tegra-ubuntu:~$ dmesg | grep video
[    0.517849] Linux video capture interface: v2.00
[    4.190036] usbcore: registered new interface driver uvcvideo
[1202341.221087] uvcvideo: Found UVC 1.00 device <unnamed> (046d:0802)
ubuntu@tegra-ubuntu:~$
```

Figure 2: Verification of UVC Driver loaded

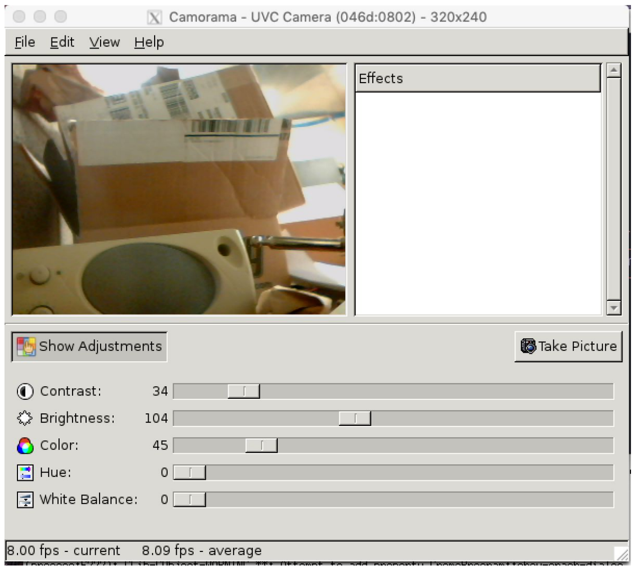
Problem 2

[10 points] Option 1: Camorama

If you do not have ***camorama***, do ***apt-get install camorama*** on your DE1-SoC, Raspberry Pi, or Jetson board [you may need to first do ***sudo add-apt-repository universe; sudo apt-get update***]. 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/). Running camorama should provide an interactive camera control session for your Logitech C2xx camera – if you have issues connecting to your camera do a "i" 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 screenshot of the tool and provide both in your report.

Solution:

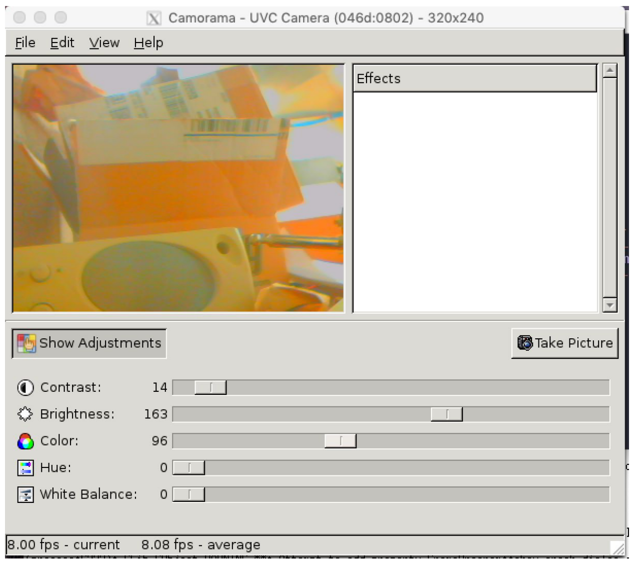
We installed camorama onto our Jetson TK1 Board using ***sudo add-apt-repository universe; sudo apt-get update***; ***sudo apt-get install camorama.*** Camorama is a GNOME2 Video4Linux Viewer which essentially just makes it easier to view, record and add effects to video streams on Linux based systems. The following figures illustrate us being able to successfully run this program and also change the effects on the live video stream.



Contrast: 34
Brightness: 104
Color: 45
Hue: 0
White Balance: 0

8.00 fps - current 8.09 fps - average

Figure 3: Verification of Camorama running successfully on our Jetson



Contrast: 14
Brightness: 163
Color: 96
Hue: 0
White Balance: 0

8.00 fps - current 8.08 fps - average

Figure 4:: Verification that changing effects in Camorama actually changes stream

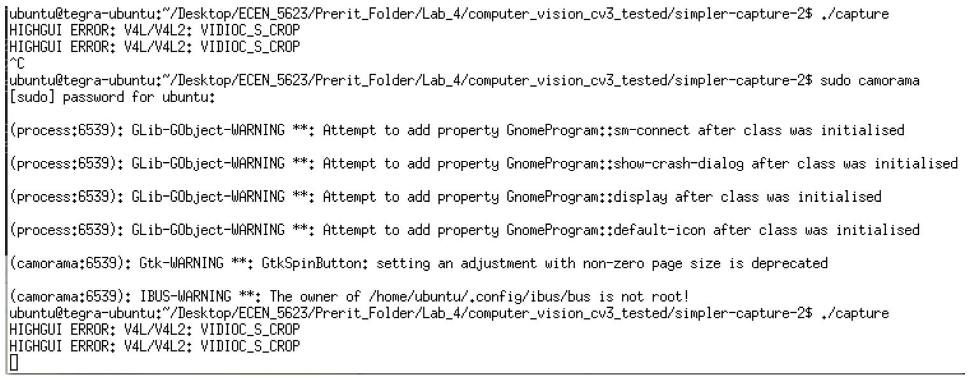
---

Problem 3

[10 points] Using your verified Logitech C200 camera on a DE1-SoC, Raspberry Pi or Jetson, verify that it can stream continuously using to a raw image buffer for transformation and processing using example code from the computer-vision or computer_vision_cv3_tested folder such as simple-capture,simpler-capture, or simpler-capture-2. Read the code and modify the device that is opened if necessary to get this to work. Provide a screen shot to prove that you got continuous capture to work.

Solution:

Our Jetson TK1 board came preinstalled with the OpenCV libraries and so we didn't have to install any of those libraries as per the provided instructions and were able to build and test the example code from the computer_vision_cv2_tested folder pretty easily. The following in an output of running simpler-capture:



```
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Prerit_Folder/Lab_4/computer_vision_cv3_tested/simpler-capture-2$ ./capture
HIGHGUI ERROR: V4L/V4L2: VIDIOC_S_CROP
HIGHGUI ERROR: V4L/V4L2: VIDIOC_S_CROP
^C
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Prerit_Folder/Lab_4/computer_vision_cv3_tested/simpler-capture-2$ sudo camorama
[sudo] password for ubuntu:

(process:6539): GLib-GObject-WARNING **: Attempt to add property GnomeProgram::sm-connect after class was initialised

(process:6539): GLib-GObject-WARNING **: Attempt to add property GnomeProgram::show-crash-dialog after class was initialised

(process:6539): GLib-GObject-WARNING **: Attempt to add property GnomeProgram::display after class was initialised

(process:6539): GLib-GObject-WARNING **: Attempt to add property GnomeProgram::default-icon after class was initialised

(camorama:6539): Gtk-WARNING **: GtkSpinButton: setting an adjustment with non-zero page size is deprecated

(camorama:6539): IBUS-WARNING **: The owner of /home/ubuntu/.config/ibus/bus is not root!
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Prerit_Folder/Lab_4/computer_vision_cv3_tested/simpler-capture-2$ ./capture
HIGHGUI ERROR: V4L/V4L2: VIDIOC_S_CROP
HIGHGUI ERROR: V4L/V4L2: VIDIOC_S_CROP
```

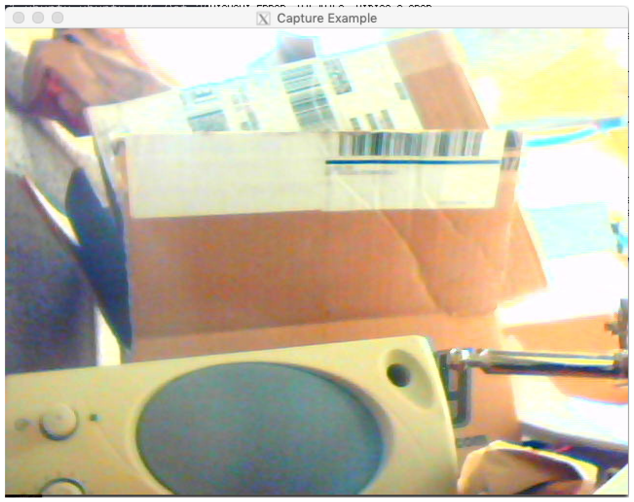
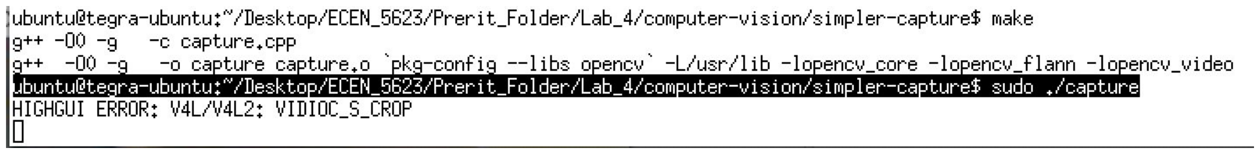


Figure 5: Verification of running simpler-capture

The following is an output for building and running simpler-capture in the computer-vision folder.



```
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Prerit_Folder/Lab_4/computer-vision/simpler-capture$ make
g++ -O0 -g   -c capture.cpp
g++  -O0 -g   -o capture capture.o `pkg-config --libs opencv` -L/usr/lib -lopencv_core -lopencv_flann -lopencv_video
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Prerit_Folder/Lab_4/computer-vision/simpler-capture$ sudo ./capture
HIGHGUI ERROR: V4L/V4L2: VIDIOC_S_CROP
▯
```

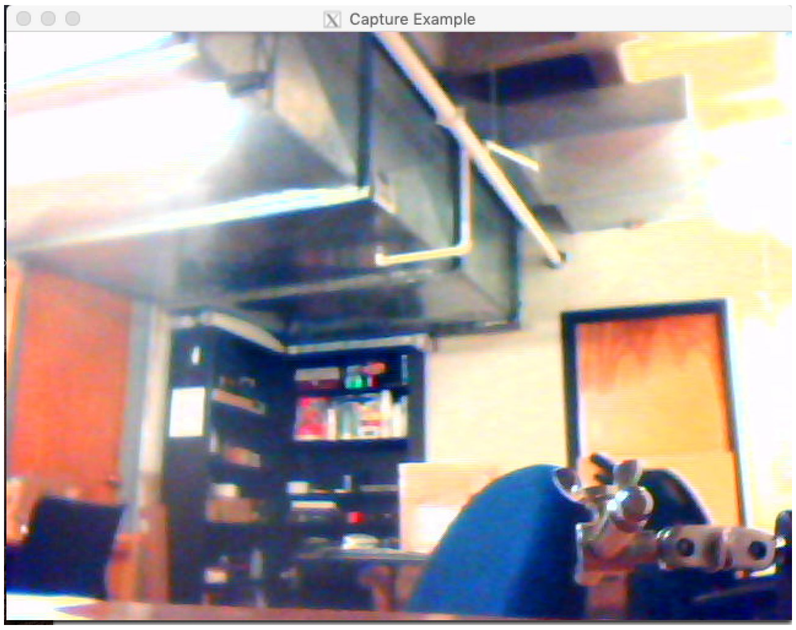


Figure 6: Verification of running simpler-capture in Computer Vision folder

----

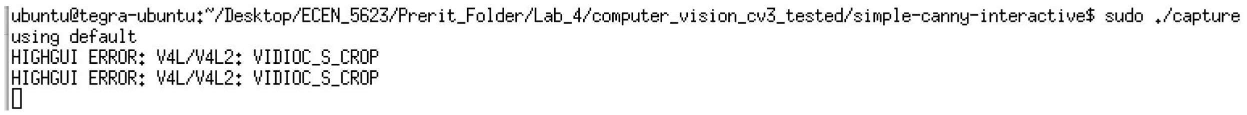
Problem 4

[20 points] Choose a continuous transformation OpenCV example from computer-vision such as the canny-interactive,hough-interactive,hough-eliptical-interactive, or stereo-transform-impoved or the from the same 4 transforms in computer vision cv3 tested. Show a screenshot to prove you built and ran the code. Provide a detailed explanation of the code and research uses for the continuous transformation by looking up API functions in the OpenCV manual (http://docs.opencv.org) and for stereo-transform-improved either implement or explain how you could make this work continuously rather than snapshot only.

Solution:

Simple Canny Interactive

The following demonstrates that we were able to build and run the simple-canny-interactive example:



```
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Prerit_Folder/Lab_4/computer_vision_cv3_tested/simple-canny-interactive$ sudo ./capture
using default
HIGHGUI ERROR: V4L/V4L2: VIDIOC_S_CROP
HIGHGUI ERROR: V4L/V4L2: VIDIOC_S_CROP
```

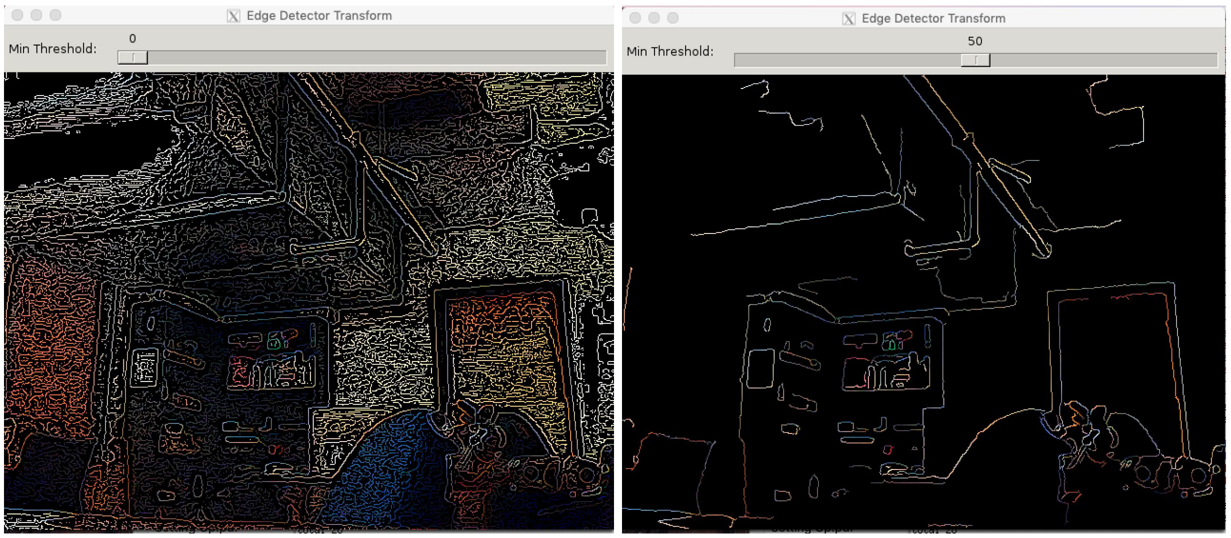
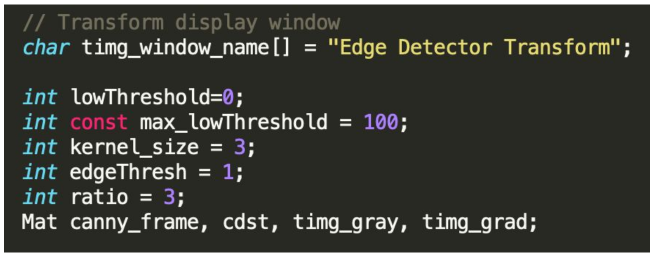


Figure 7: Verification of running simple-canny-interactive

The code is essentially doing an edge detection based on the minimum threshold specified by the interactive bar at the top.

Diving deeper into the code, it starts off by checking if the user has specified a device as an argument when calling the program. If not, it uses the default device number (i.e. 0) and continues with that device.

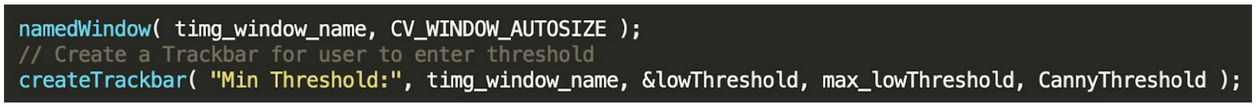
Now, the code uses global variables associated with the threshold slider, ratio and canny frames so that these references can be used in the main loop as well as when updating the edge-detection threshold.



```
// Transform display window
char timg_window_name[] = "Edge Detector Transform";

int lowThreshold=0;
int const max_lowThreshold = 100;
int kernel_size = 3;
int edgeThresh = 1;
int ratio = 3;
Mat canny_frame, cdst, timg_gray, timg_grad;
```

Figure 8: Use of global variables in simple-canny-interactive

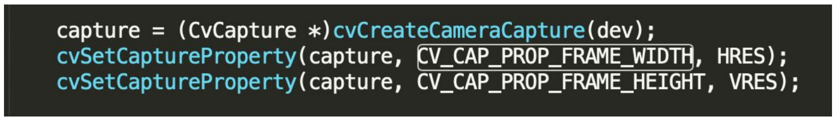
Then, in the main loop of the actual program, it creates an OpenCV window used as a placeholder for the images and trackbar with the specified name and automatic window size using the *namedWindow()* function. Upon successfully creating the window, it creates a trackbar in the same window with the *createTrackbar()* function which provides references to the declared global variables to update as well as a reference to the function to call when changing the value of the trackbar.



```
namedWindow( timg_window_name, CV_WINDOW_AUTOSIZE );
// Create a Trackbar for user to enter threshold
createTrackbar( "Min Threshold:", timg_window_name, &lowThreshold, max_lowThreshold, CannyThreshold );
```

Figure 9: Use of namedWindow() and createTrackbar() in simple-canny-interactive

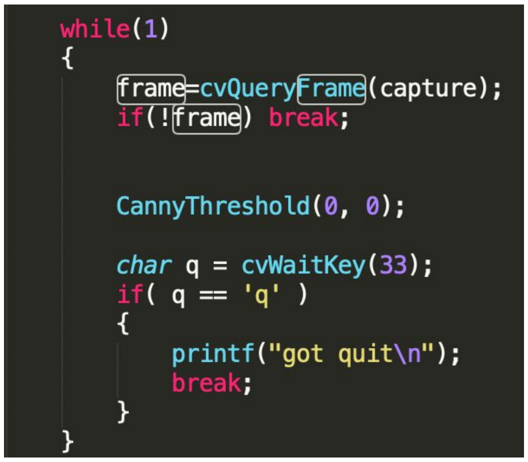
An image is then initially captured from the device specified (default is device 0) using the cvCreateCameraCapture function and the information is stored in a data structure. The properties of the data structure are updated to reflect the given height and width resolution (which in this case is 640x480.



```
    capture = (CvCapture *)cvCreateCameraCapture(dev);
    cvSetCaptureProperty(capture, CV_CAP_PROP_FRAME_WIDTH, HRES);
    cvSetCaptureProperty(capture, CV_CAP_PROP_FRAME_HEIGHT, VRES);
```

Figure 10: Storing camera image and updating properties

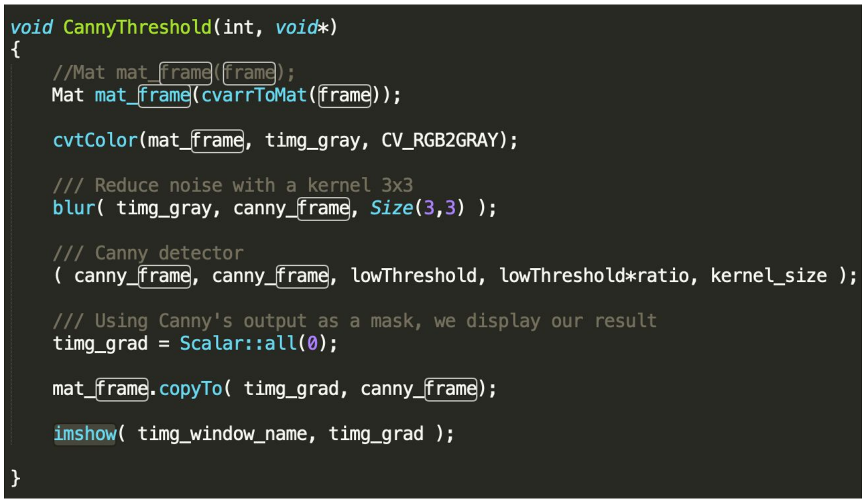
Finally, the main loop has a loop which runs until the 'q' key is pressed or the program is terminated and this loop captures a frame from the camera and stores it in an Image frame. The function CannyThreshold is then called with the parameters 0 and 0 as the min threshold values.



```
while(1)
{
    frame=cvQueryFrame(capture);
    if(!frame) break;


    CannyThreshold(0, 0);

    char q = cvWaitKey(33);
    if( q == 'q' )
    {
        printf("got quit\n");
        break;
    }
}
```

Figure 11: Loop which is continuously capturing the frames from camera

The CannyThreshold function is responsible for converting the CV array (stored in the image frame pointer) to a Matrix through the cvarrToMat(frame) function, converting the image from one color to another (timg_gray -> CV_RGB2GRAY), reducing the noise of the frame and then calling the canny function. The Canny function from the OpenCV api is a used to implement the canny edge detector which takes in the threshold, the updated kernel_size, and the detected frame images. The timing_grad is filled with all zeros which means that the image is essentially all black and then after copying the canny matrix to timing_grad, we are able to see the canny image as an output.

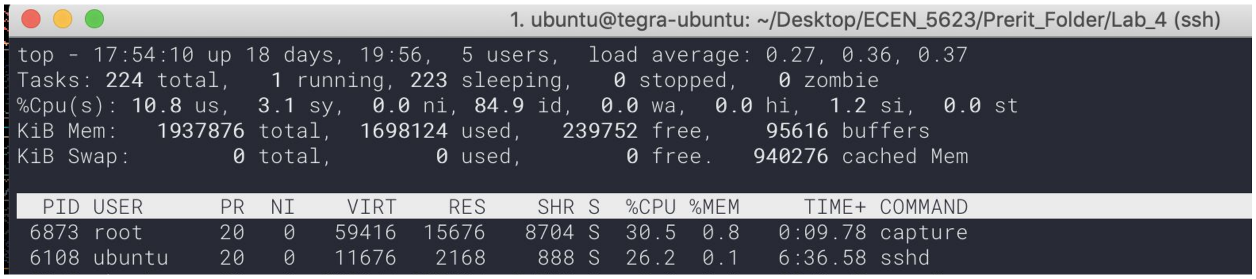


```
void CannyThreshold(int, void*)
{
    //Mat mat_frame(frame);
    Mat mat_frame(cvarrToMat(frame));

    cvtColor(mat_frame, timg_gray, CV_RGB2GRAY);

    /// Reduce noise with a kernel 3x3
    blur( timg_gray, canny_frame, Size(3,3) );

    /// Canny detector
    ( canny_frame, canny_frame, lowThreshold, lowThreshold*ratio, kernel_size );

    /// Using Canny's output as a mask, we display our result
    timg_grad = Scalar::all(0);

    mat_frame.copyTo( timg_grad, canny_frame);

    imshow( timg_window_name, timg_grad );

}
```

Figure 12: Canny Threshold Function

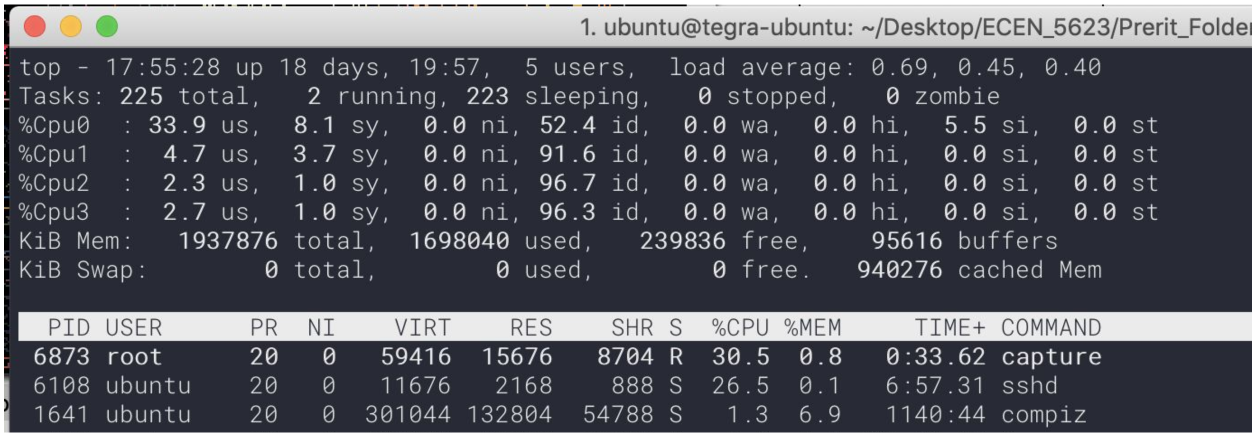
Simple Canny Interactive - CPU Loading

So with the given resolution of 640x480 as HRES and VRES, we saw the CPU load to be in the range of 30.5% as shown in the following figures.



1. ubuntu@tegra-ubuntu: ~/Desktop/ECEN_5623/Prerit_Folder/Lab_4 (ssh)

```
top - 17:54:10 up 18 days, 19:56,  5 users,  load average: 0.27, 0.36, 0.37
Tasks: 224 total,   1 running, 223 sleeping,   0 stopped,   0 zombie
%Cpu(s): 10.8 us,  3.1 sy,  0.0 ni, 84.9 id,  0.0 wa,  0.0 hi,  1.2 si,  0.0 st
KiB Mem:   1937876 total,  1698124 used,   239752 free,    95616 buffers
KiB Swap:        0 total,        0 used,        0 free.   940276 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 6873 root      20   0   59416  15676   8704 S  30.5  0.8   0:09.78 capture
 6108 ubuntu    20   0   11676   2168    888 S  26.2  0.1   6:36.58 sshd
```



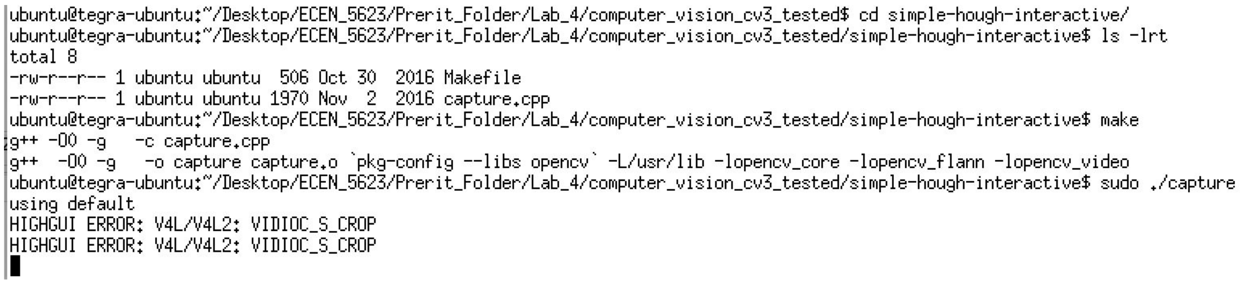
1. ubuntu@tegra-ubuntu: ~/Desktop/ECEN_5623/Prerit_Folde

```
top - 17:55:28 up 18 days, 19:57,  5 users,  load average: 0.69, 0.45, 0.40
Tasks: 225 total,   2 running, 223 sleeping,   0 stopped,   0 zombie
%Cpu0  : 33.9 us,  8.1 sy,  0.0 ni, 52.4 id,  0.0 wa,  0.0 hi,  5.5 si,  0.0 st
%Cpu1  :  4.7 us,  3.7 sy,  0.0 ni, 91.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :  2.3 us,  1.0 sy,  0.0 ni, 96.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :  2.7 us,  1.0 sy,  0.0 ni, 96.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   1937876 total,  1698040 used,   239836 free,    95616 buffers
KiB Swap:        0 total,        0 used,        0 free.   940276 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 6873 root      20   0   59416  15676   8704 R  30.5  0.8   0:33.62 capture
 6108 ubuntu    20   0   11676   2168    888 S  26.5  0.1   6:57.31 sshd
 1641 ubuntu    20   0  301044 132804  54788 S   1.3  6.9 1140:44 compiz
```

Figure 13: CPU Loading for 640x480 resolution on Canny Interactive

Simple Hough interactive

The following demonstrates that we were able to build and run the simple-hough-interactive example:



```
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Prerit_Folder/Lab_4/computer_vision_cv3_tested$ cd simple-hough-interactive/
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Prerit_Folder/Lab_4/computer_vision_cv3_tested/simple-hough-interactive$ ls -lrt
total 8
-rw-r--r-- 1 ubuntu ubuntu  506 Oct 30  2016 Makefile
-rw-r--r-- 1 ubuntu ubuntu 1970 Nov  2  2016 capture.cpp
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Prerit_Folder/Lab_4/computer_vision_cv3_tested/simple-hough-interactive$ make
g++ -O0 -g   -c capture.cpp
g++  -O0 -g   -o capture capture.o `pkg-config --libs opencv` -L/usr/lib -lopencv_core -lopencv_flann -lopencv_video
ubuntu@tegra-ubuntu:~/Desktop/ECEN_5623/Prerit_Folder/Lab_4/computer_vision_cv3_tested/simple-hough-interactive$ sudo ./capture
using default
HIGHGUI ERROR: V4L/V4L2: VIDIOC_S_CROP
HIGHGUI ERROR: V4L/V4L2: VIDIOC_S_CROP
█
```

Figure 14: Verification of running simple-hough-interactive

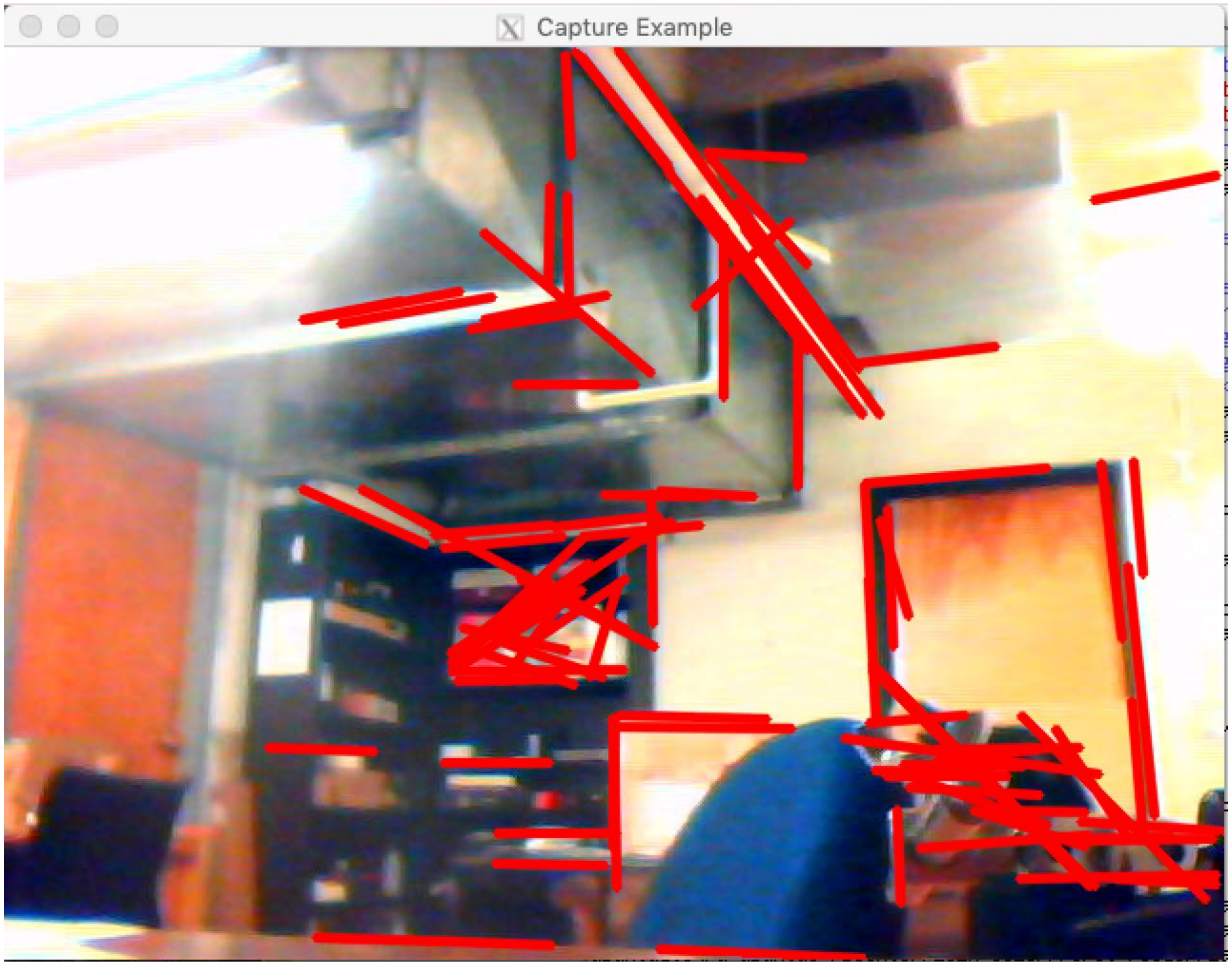
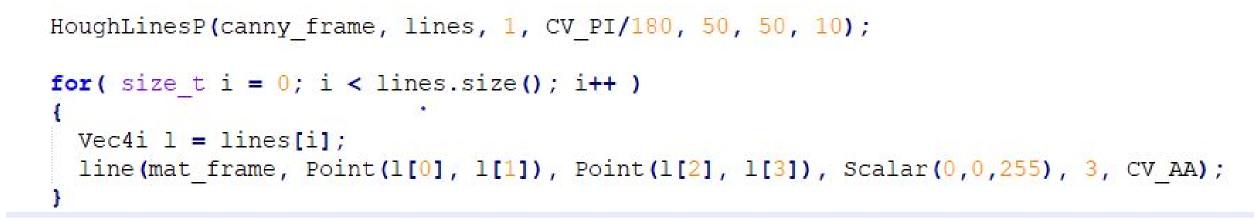


Figure 15: Output window for simple-hough-interactive

The code runs to perform hough transform on live feed of the camera attached. Hough transform is a technique used to identify lines in an image or frame.

This code is quite similar to the simple-canny-interactive. The opening of the OpenCV window, capturing of an image, setting the resolution, detect the edges, having a loop and breaking out if the key ‘q’ is pressed and close the OpenCV window while exiting.

For the Hough Transform, there is a function called ***HoughLinesP*** which detects lines as output using probabilistic theorems. The function finds line segments in the image which is converted to binary form using the *cvtColor()* function. In the function *HoughLinesP*, there are parameters which are used for defining the threshold, the rho and theta value and also the minimum line length and the maximum gap allowed between points.

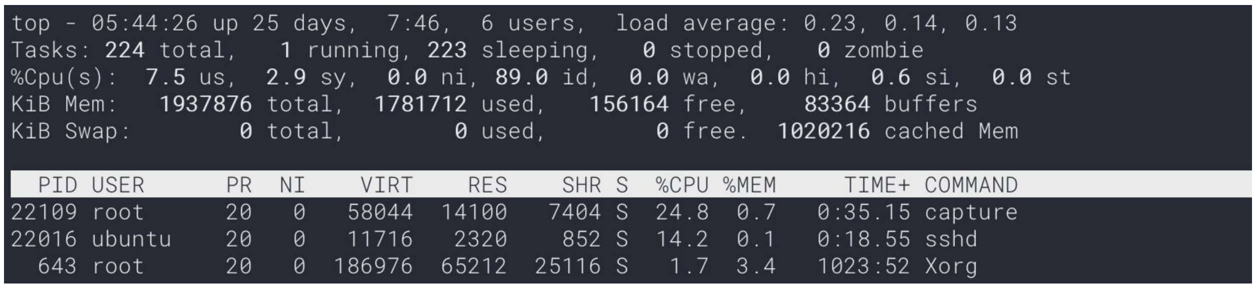


```
HoughLinesP(canny_frame, lines, 1, CV_PI/180, 50, 50, 10);

for( size_t i = 0; i < lines.size(); i++ )
{
  Vec4i l = lines[i];
  line(mat_frame, Point(l[0], l[1]), Point(l[2], l[3]), Scalar(0,0,255), 3, CV_AA);
}
```

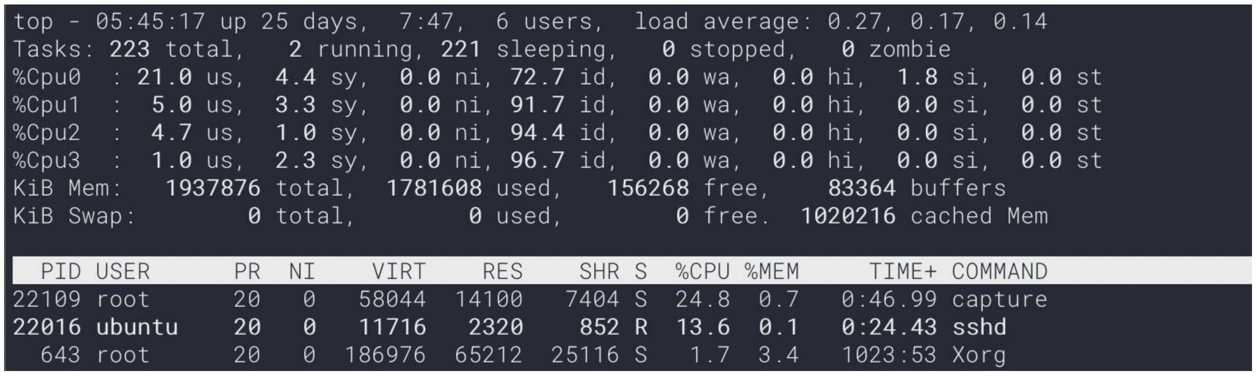
Figure 16: Main functionality for using Hough Lines Transform

The processed image after the Hough Line transformation is displayed using the *imshow* commands. We can display the lines by using the for loop where we use vectors and lines functionality to draw up the lines.

Simple Hough Interactive - CPU Loading



```
top - 05:44:26 up 25 days,  7:46,  6 users,  load average: 0.23, 0.14, 0.13
Tasks: 224 total,   1 running, 223 sleeping,   0 stopped,   0 zombie
%Cpu(s):  7.5 us,  2.9 sy,  0.0 ni, 89.0 id,  0.0 wa,  0.0 hi,  0.6 si,  0.0 st
KiB Mem:   1937876 total,  1781712 used,   156164 free,    83364 buffers
KiB Swap:        0 total,        0 used,        0 free.  1020216 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
22109 root      20   0   58044  14100   7404 S  24.8  0.7   0:35.15 capture
22016 ubuntu    20   0   11716   2320    852 S  14.2  0.1   0:18.55 sshd
  643 root      20   0  186976  65212  25116 S   1.7  3.4   1023:52 Xorg
```



```
top - 05:45:17 up 25 days,  7:47,  6 users,  load average: 0.27, 0.17, 0.14
Tasks: 223 total,   2 running, 221 sleeping,   0 stopped,   0 zombie
%Cpu0  : 21.0 us,  4.4 sy,  0.0 ni, 72.7 id,  0.0 wa,  0.0 hi,  1.8 si,  0.0 st
%Cpu1  :  5.0 us,  3.3 sy,  0.0 ni, 91.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :  4.7 us,  1.0 sy,  0.0 ni, 94.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :  1.0 us,  2.3 sy,  0.0 ni, 96.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   1937876 total,  1781608 used,   156268 free,    83364 buffers
KiB Swap:        0 total,        0 used,        0 free.  1020216 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
22109 root      20   0   58044  14100   7404 S  24.8  0.7   0:46.99 capture
22016 ubuntu    20   0   11716   2320    852 R  13.6  0.1   0:24.43 sshd
  643 root      20   0  186976  65212  25116 S   1.7  3.4   1023:53 Xorg
```

Figure 17: CPU Loading for 640x480 resolution on Hough Lines Interactive

---

Simple Hough Elliptical

The following demonstrates that we were able to build and run the simple-hough-elliptical-interactive example:

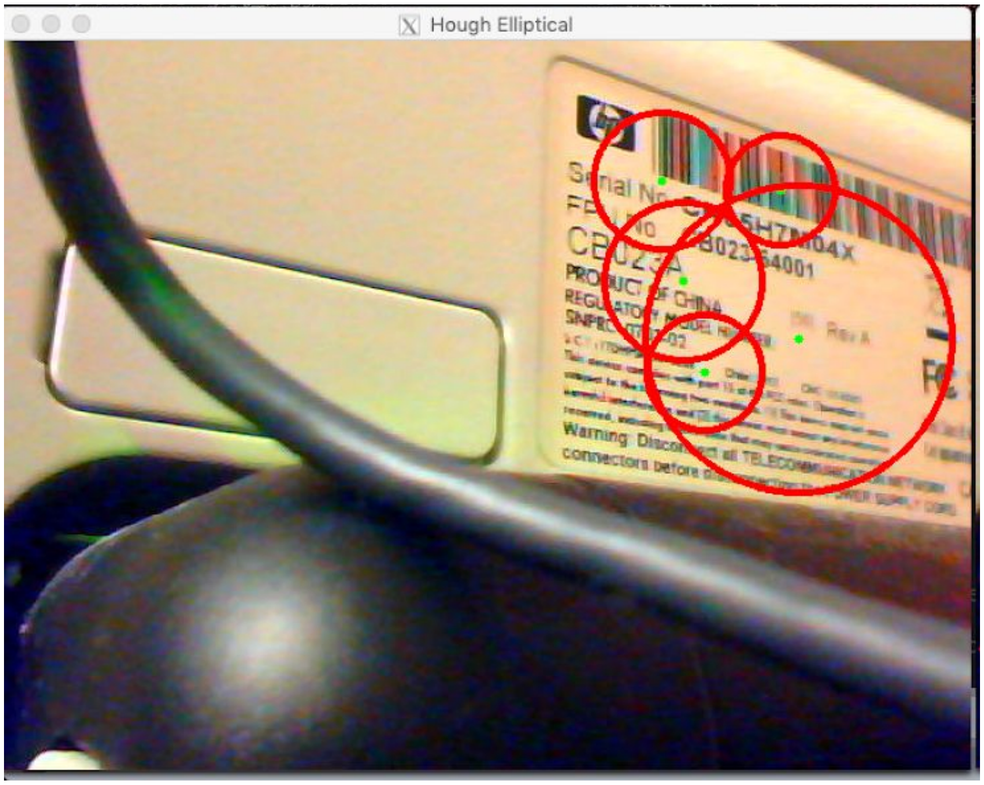
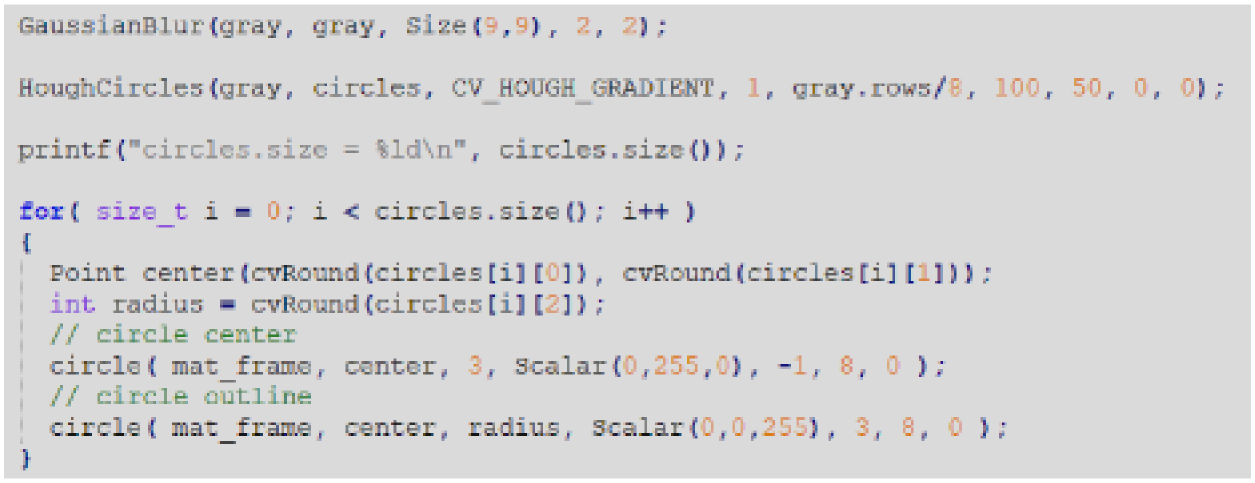


Figure 18: Verification and Output from running simple-hough-elliptical-interactive

The code uses the OpenCV function *HoughCircles()* for detecting circular objects in an image. The program uses edge detection to an blurred image and then applies Hough Circles Transform to it. The Hough Circle Transform are used to find the centers of the circles and in the next stage find the radius for that center point.

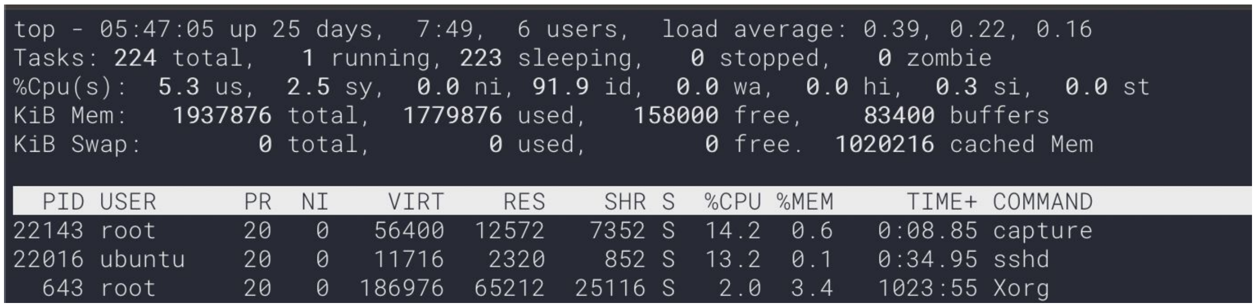
The OpenCV function *HoughCircles()* have parameters for determining the thresholds for ceter detection, the minimum and maximum radius which can be detected. Then there is a loop for drawing up all the circles and the centers in the color red for that particular loaded image.



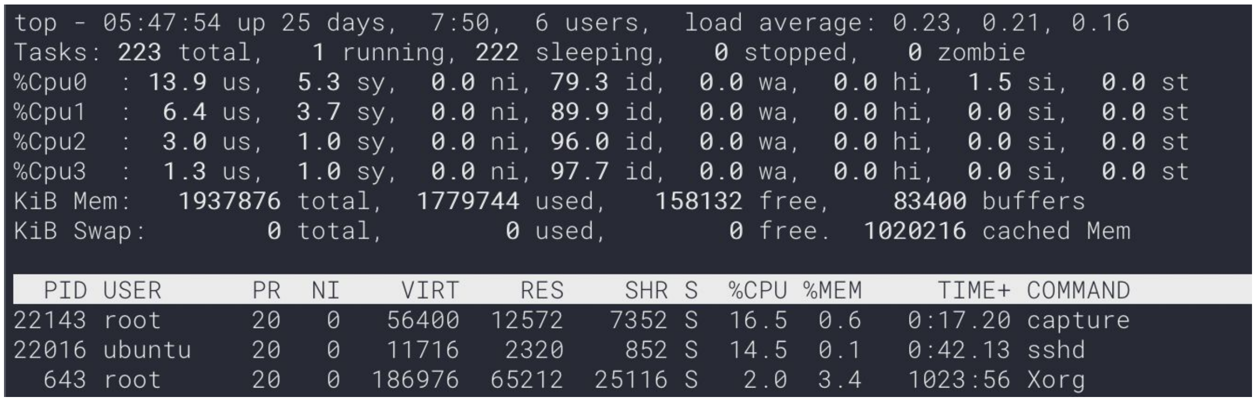
```
GaussianBlur(gray, gray, Size(9,9), 2, 2);

HoughCircles(gray, circles, CV_HOUGH_GRADIENT, 1, gray.rows/8, 100, 50, 0, 0);

printf("circles.size = %ld\n", circles.size());

for( size_t i = 0; i < circles.size(); i++ )
{
  Point center(cvRound(circles[i][0]), cvRound(circles[i][1]));
  int radius = cvRound(circles[i][2]);
  // circle center
  circle( mat_frame, center, 3, Scalar(0,255,0), -1, 8, 0 );
  // circle outline
  circle( mat_frame, center, radius, Scalar(0,0,255), 3, 8, 0 );
}
```

Figure 19: Main functionality for using Hough Circles Transform

Simple Hough Elliptical - CPU Loading



```
top - 05:47:05 up 25 days,  7:49,  6 users,  load average: 0.39, 0.22, 0.16
Tasks: 224 total,   1 running, 223 sleeping,   0 stopped,   0 zombie
%Cpu(s):  5.3 us,  2.5 sy,  0.0 ni, 91.9 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 st
KiB Mem:   1937876 total,  1779876 used,   158000 free,    83400 buffers
KiB Swap:        0 total,        0 used,        0 free.  1020216 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
22143 root      20   0   56400  12572   7352 S  14.2  0.6   0:08.85 capture
22016 ubuntu    20   0   11716   2320    852 S  13.2  0.1   0:34.95 sshd
  643 root      20   0  186976  65212  25116 S   2.0  3.4   1023:55 Xorg
```



```
top - 05:47:54 up 25 days,  7:50,  6 users,  load average: 0.23, 0.21, 0.16
Tasks: 223 total,   1 running, 222 sleeping,   0 stopped,   0 zombie
%Cpu0  : 13.9 us,  5.3 sy,  0.0 ni, 79.3 id,  0.0 wa,  0.0 hi,  1.5 si,  0.0 st
%Cpu1  :  6.4 us,  3.7 sy,  0.0 ni, 89.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :  3.0 us,  1.0 sy,  0.0 ni, 96.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :  1.3 us,  1.0 sy,  0.0 ni, 97.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   1937876 total,  1779744 used,   158132 free,    83400 buffers
KiB Swap:        0 total,        0 used,        0 free.  1020216 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
22143 root      20   0   56400  12572   7352 S  16.5  0.6   0:17.20 capture
22016 ubuntu    20   0   11716   2320    852 S  14.5  0.1   0:42.13 sshd
  643 root      20   0  186976  65212  25116 S   2.0  3.4   1023:56 Xorg
```

Figure 20: CPU Loading for 640x480 resolution on Hough Circles Interactive

----

Problem 5

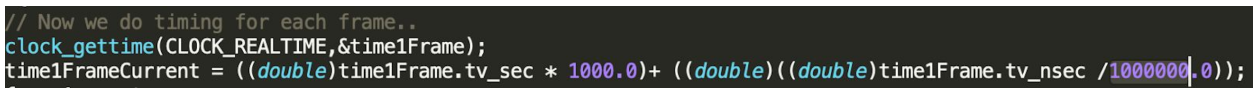
[50 points] Using a Logitech C200, choose 3 real-time interactive transformations to compare in terms of average frame rate at a given resolution for a range of at least 3 resolutions in a common aspect ratio (e.g. 4:3 for 1280x960, 640x480, 320x240, 160x120, 80x60) by adding time-stamps to analyze the potential throughput. Based on average analysis, pick a reasonable soft real-time deadline (e.g. if average frame rate is 12 Hz, choose a deadline of 100 milliseconds to provide some margin) and convert the processing to SCHED_FIFO and determine if you can meet deadlines with predictability and measure jitter in the frame rate relative to your deadline.

Solution:

For this problem, we chose to do the canny edge detector, hough line and hough elliptical transformations. In order to run all the three transformations, we implemented 3 different pthreads which would run in the userspace - primarily being synchronized by semaphores. As a consequence, we used three different semaphores, each indicating when the appropriate task should be run with the following design:

- The main loop initializes the canny thread to run first by initializing the semaphore with a value of 1 and the other semaphores with a value of 0.
- The canny thread then executed to completion for a specific number of frames (i.e. chose 50) and upon completion, pushes the value of the semaphore associated with the hough elliptical transform.
- The hough elliptical transform is then able to run for the same number of frames, the average fps, total frames and average execution of time per frame is reported and upon completion, pushes the value of the semaphore associated with the hough normal transform.
- Finally, the hough normal transformation runs and upon completion, all the threads will have finished executing allowing for these threads to be joined by the main loop/

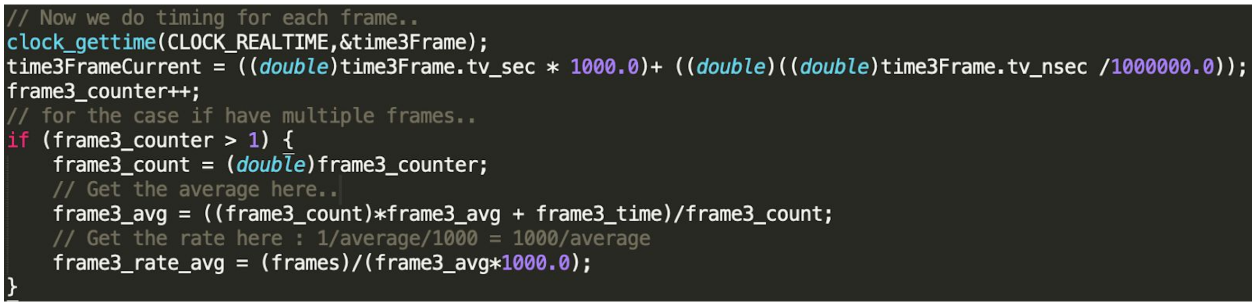
In order to conduct the algorithm analysis, we used standard timing structs as we've been doing for the previous exercises and just measured the duration of the transformation. Specifically, the first step is to get the current timestamp using the clock_gettime() function and then given that it outputs seconds and nanoseconds, we converted it to get the current time in milliseconds by converting the seconds to ms (multiplying by 1000) and the nanoseconds to ms (dividing by 1000000). This is demonstrated in the following figure:



```
// Now we do timing for each frame..
clock_gettime(CLOCK_REALTIME,&time1Frame);
time1FrameCurrent = ((double)time1Frame.tv_sec * 1000.0)+ ((double)((double)time1Frame.tv_nsec /1000000.0));
```

Figure 21: Getting the current time in ms prior to transformation execution

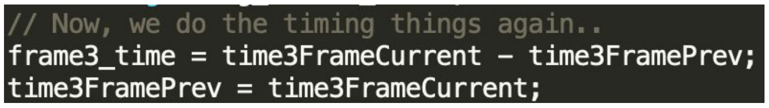
We then want to keep a running average of the multiple frames and in order to do this, we check if we have more than 1 frame. If this is the case, we get the average by taking the number of frames and multiplying that with the previous average as to scale it, add that to the execution time of the previous frame and divide by the number of frames. The rate (in fps) is then determined by taking the number of frames and dividing this by the seconds (ms*1000).



```
// Now we do timing for each frame..
clock_gettime(CLOCK_REALTIME,&time3Frame);
time3FrameCurrent = ((double)time3Frame.tv_sec * 1000.0)+ ((double)((double)time3Frame.tv_nsec /1000000.0));
frame3_counter++;
// for the case if have multiple frames..
if (frame3_counter > 1) {
    frame3_count = (double)frame3_counter;
    // Get the average here..
    frame3_avg = ((frame3_count)*frame3_avg + frame3_time)/frame3_count;
    // Get the rate here : 1/average/1000 = 1000/average
    frame3_rate_avg = (frames)/(frame3_avg*1000.0);
}
```

Figure 22: Getting the average time for multiple frames

Finally, we actually compute the appropriate transformations and all of this is derived from the examples provided by Sam Siewart's code as outlined in Problem 4. Post transformation, we update the times for the next frame as such:

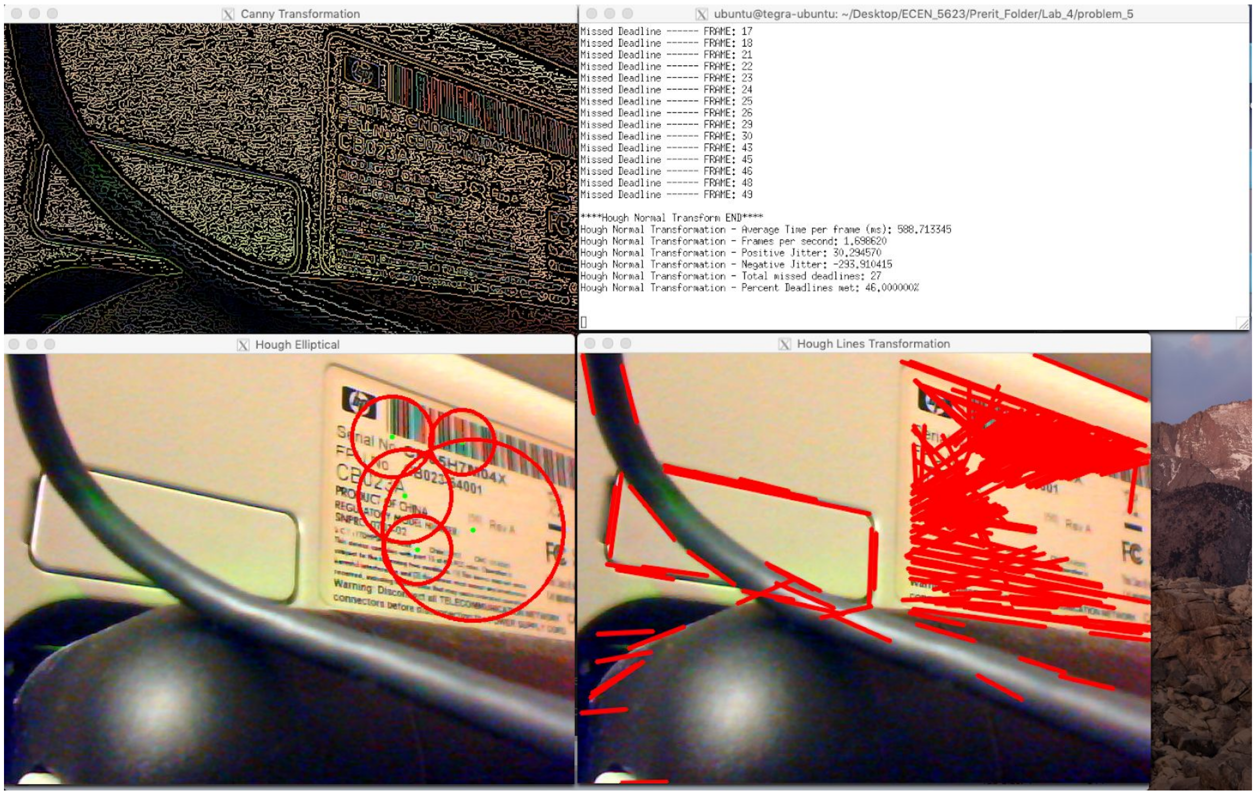


```
// Now, we do the timing things again..
frame3_time = time3FrameCurrent - time3FramePrev;
time3FramePrev = time3FrameCurrent;
```

Figure 23: Updating timing after processing of each frame

And finally, we output the results for the average execution time per frame and the frames per second.

Our prototype analysis outputs the following results demonstrating that it is working correctly.


Canny Transformation
ubuntu@tegra-ubuntu: ~/Desktop/ECEN_5623/Prerit_Folder/Lab_4/problem_5
Hough Elliptical
Hough Lines Transformation

```
Missed Deadline ------ FRAME: 17
Missed Deadline ------ FRAME: 18
Missed Deadline ------ FRAME: 21
Missed Deadline ------ FRAME: 22
Missed Deadline ------ FRAME: 23
Missed Deadline ------ FRAME: 24
Missed Deadline ------ FRAME: 25
Missed Deadline ------ FRAME: 26
Missed Deadline ------ FRAME: 29
Missed Deadline ------ FRAME: 30
Missed Deadline ------ FRAME: 43
Missed Deadline ------ FRAME: 45
Missed Deadline ------ FRAME: 46
Missed Deadline ------ FRAME: 48
Missed Deadline ------ FRAME: 49

****Hough Normal Transform END****
Hough Normal Transformation - Average Time per frame (ms): 588.713345
Hough Normal Transformation - Frames per second: 1.698620
Hough Normal Transformation - Positive Jitter: 30.294570
Hough Normal Transformation - Negative Jitter: -293.910415
Hough Normal Transformation - Total missed deadlines: 27
Hough Normal Transformation - Percent Deadlines met: 46.000000%
```

Figure 24: Output of #5

Using this methodology, we were able to arrive at the following results for the 3 different examples and different resolutions for the algorithm analysis:

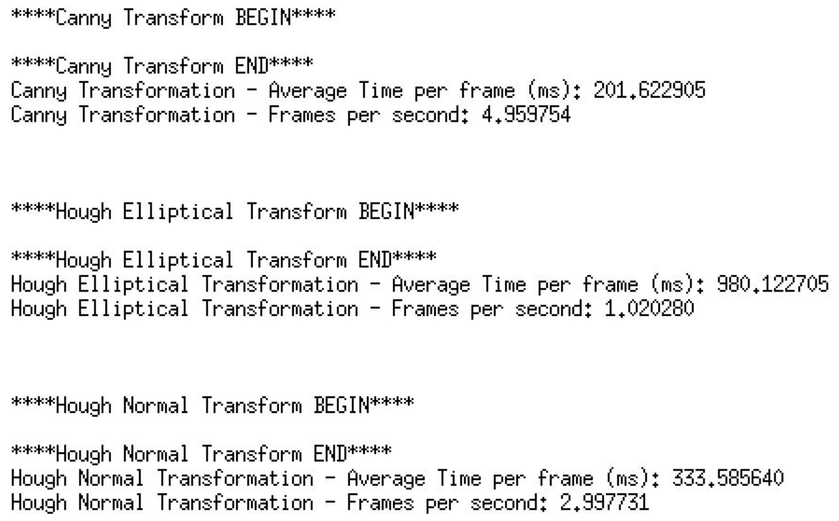
80x60 Resolution:

```
****Canny Transform BEGIN****

****Canny Transform END****
Canny Transformation - Average Time per frame (ms): 173.529409
Canny Transformation - Frames per second: 5.762712
```

```
****Hough Elliptical Transform BEGIN****

****Hough Elliptical Transform END****
Hough Elliptical Transformation - Average Time per frame (ms): 951.971050
Hough Elliptical Transformation - Frames per second: 1.050452
```

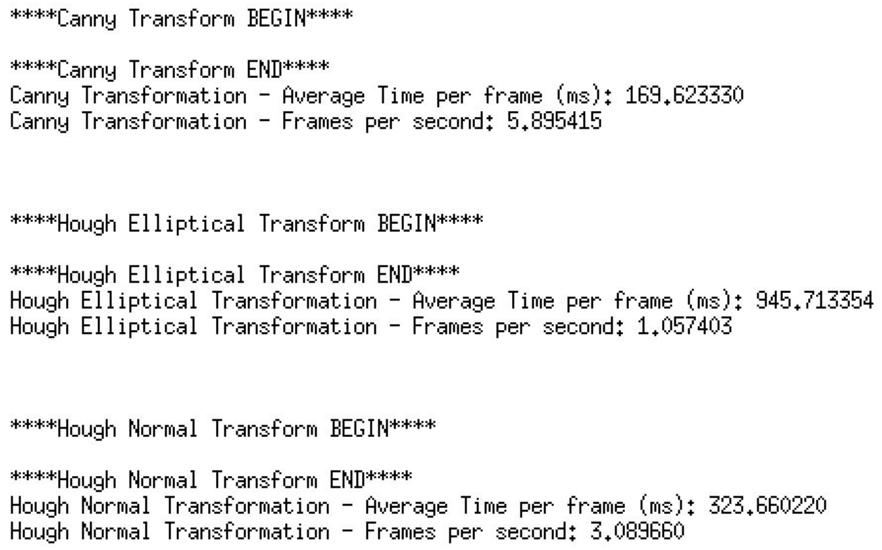
```
****Hough Normal Transform BEGIN****

****Hough Normal Transform END****
Hough Normal Transformation - Average Time per frame (ms): 297.038237
Hough Normal Transformation - Frames per second: 3.366570
```

160x120 Resolution:



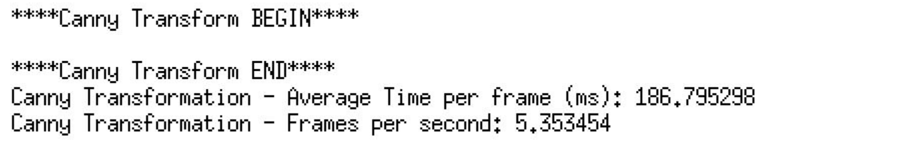
```
****Canny Transform BEGIN****

****Canny Transform END****
Canny Transformation - Average Time per frame (ms): 201.622905
Canny Transformation - Frames per second: 4.959754



****Hough Elliptical Transform BEGIN****

****Hough Elliptical Transform END****
Hough Elliptical Transformation - Average Time per frame (ms): 980.122705
Hough Elliptical Transformation - Frames per second: 1.020280



****Hough Normal Transform BEGIN****

****Hough Normal Transform END****
Hough Normal Transformation - Average Time per frame (ms): 333.585640
Hough Normal Transformation - Frames per second: 2.997731
```

320x240 Resolution:

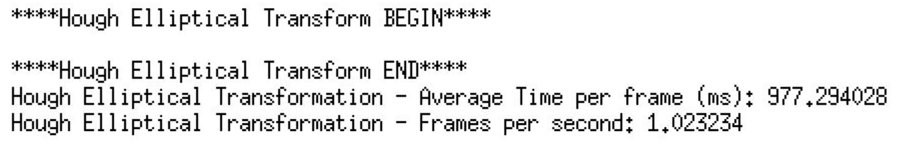


```
****Canny Transform BEGIN****

****Canny Transform END****
Canny Transformation - Average Time per frame (ms): 169.623330
Canny Transformation - Frames per second: 5.895415



****Hough Elliptical Transform BEGIN****

****Hough Elliptical Transform END****
Hough Elliptical Transformation - Average Time per frame (ms): 945.713354
Hough Elliptical Transformation - Frames per second: 1.057403



****Hough Normal Transform BEGIN****

****Hough Normal Transform END****
Hough Normal Transformation - Average Time per frame (ms): 323.660220
Hough Normal Transformation - Frames per second: 3.089660
```

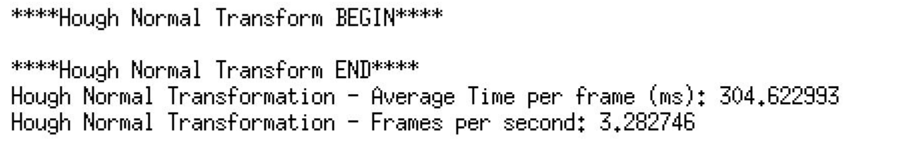
640x480 Resolution:



```
****Canny Transform BEGIN****

****Canny Transform END****
Canny Transformation - Average Time per frame (ms): 186.795298
Canny Transformation - Frames per second: 5.353454
```

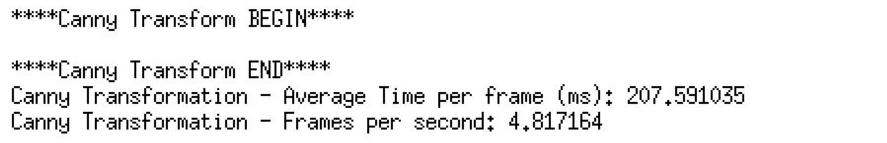


```
****Hough Elliptical Transform BEGIN****

****Hough Elliptical Transform END****
Hough Elliptical Transformation - Average Time per frame (ms): 977.294028
Hough Elliptical Transformation - Frames per second: 1.023234
```

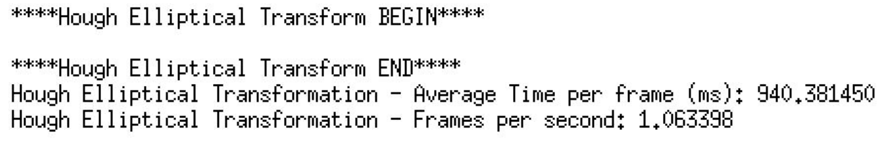


```
****Hough Normal Transform BEGIN****

****Hough Normal Transform END****
Hough Normal Transformation - Average Time per frame (ms): 304.622993
Hough Normal Transformation - Frames per second: 3.282746
```

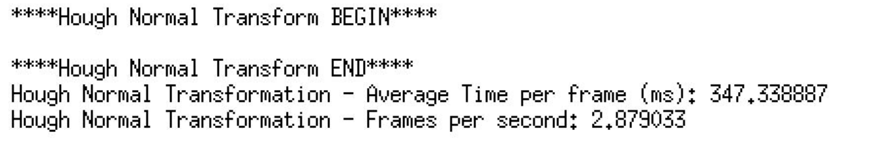
1280x960 Resolution:



```
****Canny Transform BEGIN****

****Canny Transform END****
Canny Transformation - Average Time per frame (ms): 207.591035
Canny Transformation - Frames per second: 4.817164
```



```
****Hough Elliptical Transform BEGIN****

****Hough Elliptical Transform END****
Hough Elliptical Transformation - Average Time per frame (ms): 940.381450
Hough Elliptical Transformation - Frames per second: 1.063398
```



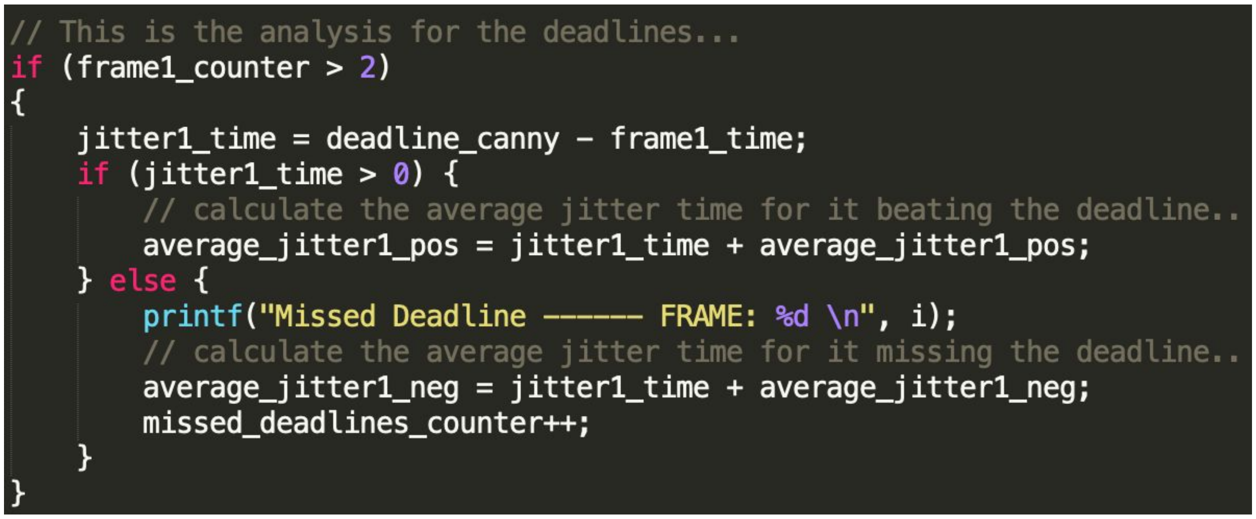
```
****Hough Normal Transform BEGIN****

****Hough Normal Transform END****
Hough Normal Transformation - Average Time per frame (ms): 347.338887
Hough Normal Transformation - Frames per second: 2.879033
```

Therefore, from the results above, it can be seen that as the resolution differs, so does the frames per second as well as the average time of execution per frame. This drives the motivation to choose different soft-real time deadlines for each process based on the resolution as well as per particular thread because each thread takes a different amount of time to execute. As a result, we will continue with the standard 640x480 resolution and so therefore, we will use that as the baseline for all the transformation jitter calculations.

The deadlines can be derived from the frames per second as outputted in figures above. Specifically, note that if we have a frame rate of 12 Hz, a suitable deadline would be 100 ms because 1/12 = 0.083 and therefore also accounts for a bit of margin. In our case, we have the following soft-realtime deadlines for the 640x480 resolution:

Canny Transformation: fps = 5.35 and so 1/(5.35) = 0.187 ~ 200 ms
Hough Elliptical: fps = 1.023 and so 1/(1.023) = 0.977 ~ 1300 ms
Hough Transform: fps = 3.282 and so 1/(3.282) = 0.304 ~ 350 ms

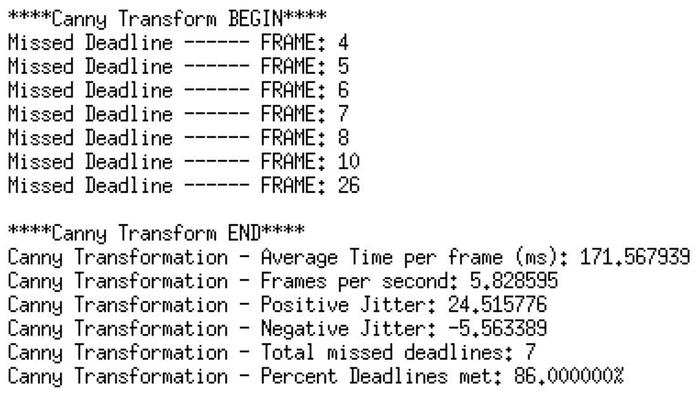
Using these derived deadlines, we can calculate the jitter as it is simply the execution time subtracted from the deadline for the process. This number can either be positive (i.e. the execution time is faster than the deadline) or negative (slower than the deadline) in which case it misses the deadline for the frame. Implementing this in the code, we have the following logic:



```
// This is the analysis for the deadlines...
if (frame1_counter > 2)
{
    jitter1_time = deadline_canny - frame1_time;
    if (jitter1_time > 0) {
        // calculate the average jitter time for it beating the deadline..
        average_jitter1_pos = jitter1_time + average_jitter1_pos;
    } else {
        printf("Missed Deadline ------ FRAME: %d \n", i);
        // calculate the average jitter time for it missing the deadline..
        average_jitter1_neg = jitter1_time + average_jitter1_neg;
        missed_deadlines_counter++;
    }
}
```

Figure 25: Jitter Analysis

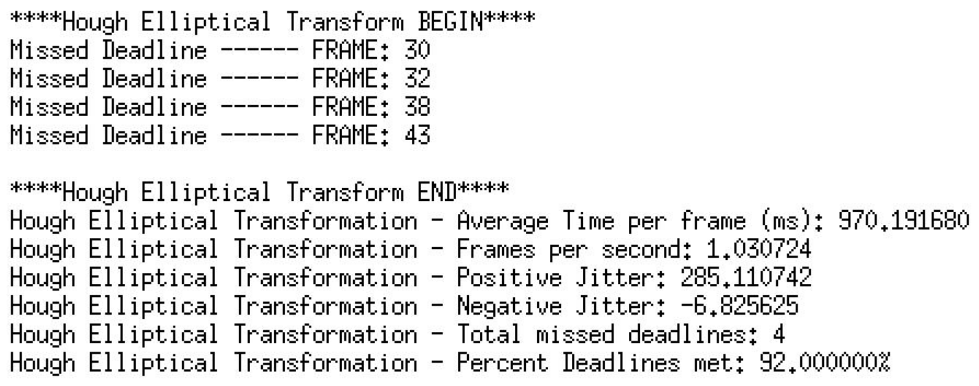
As a consequence, seeing this in practice for the canny transformation, we get the following results for 640x480 resolution:



```
****Canny Transform BEGIN****
Missed Deadline ------ FRAME: 4
Missed Deadline ------ FRAME: 5
Missed Deadline ------ FRAME: 6
Missed Deadline ------ FRAME: 7
Missed Deadline ------ FRAME: 8
Missed Deadline ------ FRAME: 10
Missed Deadline ------ FRAME: 26

****Canny Transform END****
Canny Transformation - Average Time per frame (ms): 171.567939
Canny Transformation - Frames per second: 5.828595
Canny Transformation - Positive Jitter: 24.515776
Canny Transformation - Negative Jitter: -5.563389
Canny Transformation - Total missed deadlines: 7
Canny Transformation - Percent Deadlines met: 86.000000%
```

Observe that although a few deadlines were missed, we still had a 86% of the deadlines met with the average jitter ranging from -5 to 24 ms which isn't too large of a deviation.

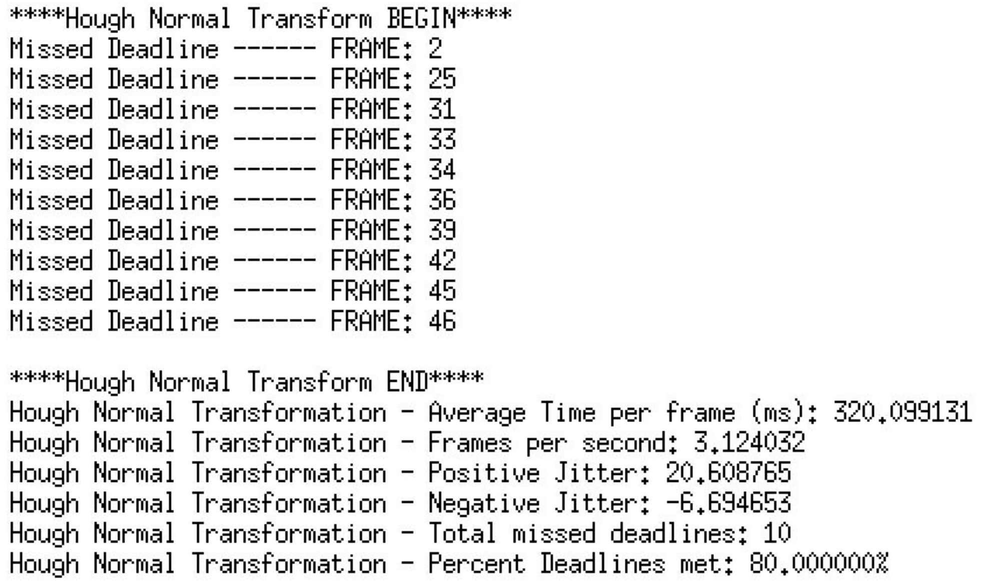
For the hough elliptical thread with a deadline of 1300 ms, we get the following results:



```
****Hough Elliptical Transform BEGIN****
Missed Deadline ------ FRAME: 30
Missed Deadline ------ FRAME: 32
Missed Deadline ------ FRAME: 38
Missed Deadline ------ FRAME: 43

****Hough Elliptical Transform END****
Hough Elliptical Transformation - Average Time per frame (ms): 970.191680
Hough Elliptical Transformation - Frames per second: 1.030724
Hough Elliptical Transformation - Positive Jitter: 285.110742
Hough Elliptical Transformation - Negative Jitter: -6.825625
Hough Elliptical Transformation - Total missed deadlines: 4
Hough Elliptical Transformation - Percent Deadlines met: 92.000000%
```

Examining the result obtained for hough elliptical, we see that lesser deadlines are missed (8%) and the jitter ranges from -6 to 285 ms.

And finally, for the hough normal transformation with a deadline of 350 ms, we get:



```
****Hough Normal Transform BEGIN****
Missed Deadline ------ FRAME: 2
Missed Deadline ------ FRAME: 25
Missed Deadline ------ FRAME: 31
Missed Deadline ------ FRAME: 33
Missed Deadline ------ FRAME: 34
Missed Deadline ------ FRAME: 36
Missed Deadline ------ FRAME: 39
Missed Deadline ------ FRAME: 42
Missed Deadline ------ FRAME: 45
Missed Deadline ------ FRAME: 46

****Hough Normal Transform END****
Hough Normal Transformation - Average Time per frame (ms): 320.099131
Hough Normal Transformation - Frames per second: 3.124032
Hough Normal Transformation - Positive Jitter: 20.608765
Hough Normal Transformation - Negative Jitter: -6.694653
Hough Normal Transformation - Total missed deadlines: 10
Hough Normal Transformation - Percent Deadlines met: 80.000000%
```

Inspecting the above result, we find that about 80% deadlines are met and the jitter are ranging from -6 to 20 ms which is very practical.

References

[1] https://docs.opencv.org/2.4/modules/highgui/doc/user_interface.html?highlight=namedwindow
[2] http://www.ideasonboard.org/uvc/
[3] https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html
[4] https://docs.opencv.org/3.4/da/d5c/tutorial_canny_detector.html
[5] https://docs.opencv.org/3.4/d4/d70/tutorial_hough_circle.html
[6] https://lwn.net/Articles/203924/
[7] http://mercury.pr.erau.edu/~siewerts/cs415/code/computer-vision/
[8] http://mercury.pr.erau.edu/~siewerts/cs415/code/computer_vision_cv3_tested/
[9] https://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html