EE 146
(Computer Vision)

- Final Project Report -
Obstacle Avoidance
(Edge Detection Technique)

By:
Leo G. Ortega III (SID: 861274462)
Jason Weiser (SID: 862113384)

March 12, 2020

1 of 3 people are distracted while driving for an average of 4.6 seconds. According to the Association for Safe International Road Travel (A.S.I.R.T.), more than 90 people die everyday in car collisions. This is significant because many crashes, collisions or casualties can be avoided by installing obstacle avoidance in making vehicles. (Autonomous) Robots or cars can detect different kinds of obstacles and avoid crashes &/or casualties in mobile robots/vehicles. Thus making it more protective for its occupants and the surrounding people. Obstacle avoidance is one of the most important aspects of mobile robotics especially in emergency rescues which are impossible for humans to reach directly and require robots to detect and avoid obstacles. With vehicles that avoid obstacles, it makes robots/cars more protective for its occupants and the surrounding people. Most importantly, emergency rescues are possible for which the cases humans cannot reach to the certain point directly.

To get to the same task, Researchers use many different techniques. Researchers, like Martinsanz, projected light patterns and their corresponding distortions against a background to identify objects (Martinsanz). Another researcher has used a method called optical flow (Souhila & Karim). In optical flow, it uses the motion of the observer through an environment to create changes in the camera and move the camera slightly from side to side to create depth (as seen in Figure 1).

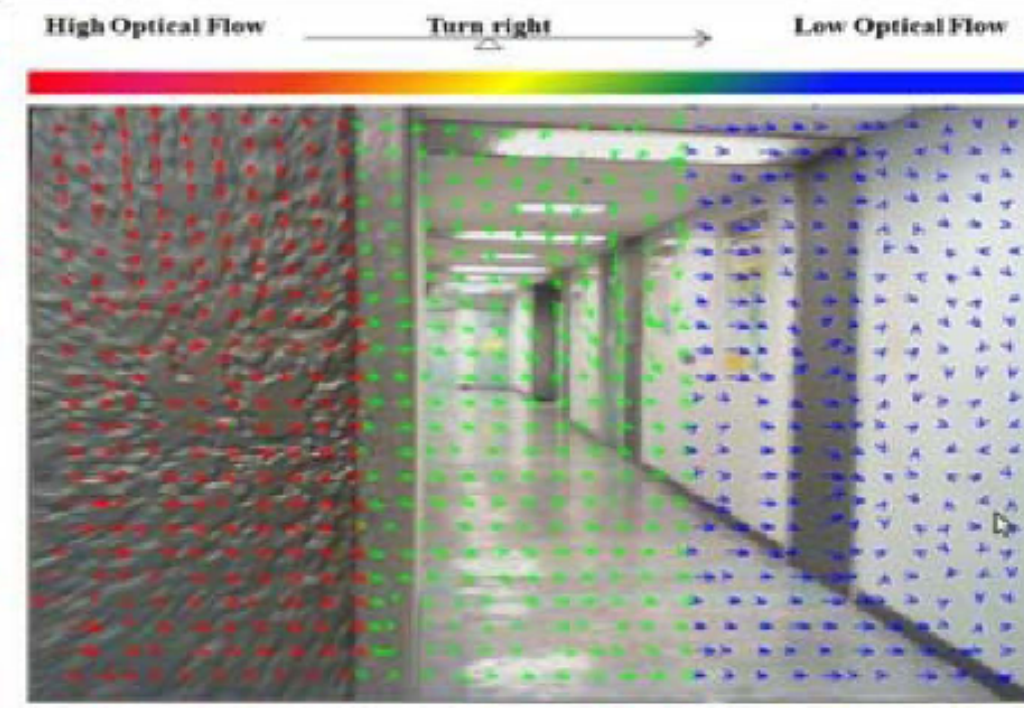


Figure 1 - Optical Flow example

The third technique, Blob Based Technique, exploits the fact that the floor is a single large object and segments the image into a smaller number of colors in order to connect pixels into blobs that we can process as objects as seen in Figure 2(Heisele & Ritter).

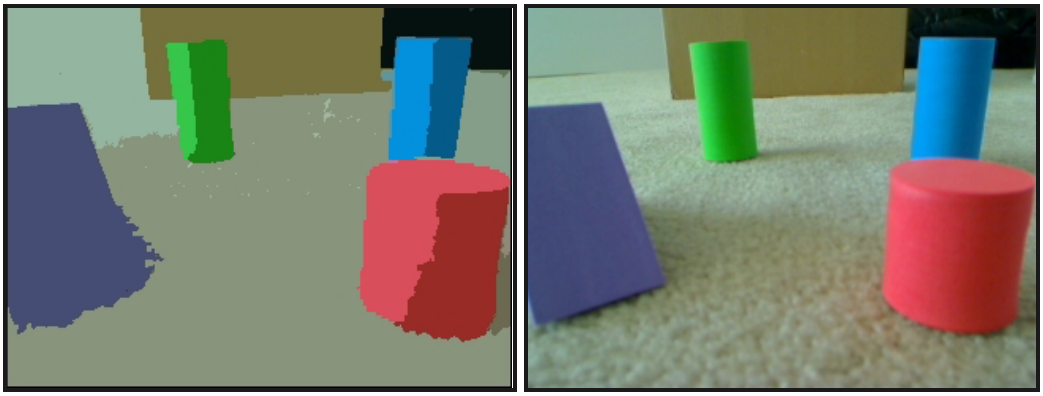


Figure 2 - Original Image, right; Blob Based Technique, left

However, the blob will pick up on the small patterns of the carpet and incorrectly see them as obstacles (as seen in Figure 3).

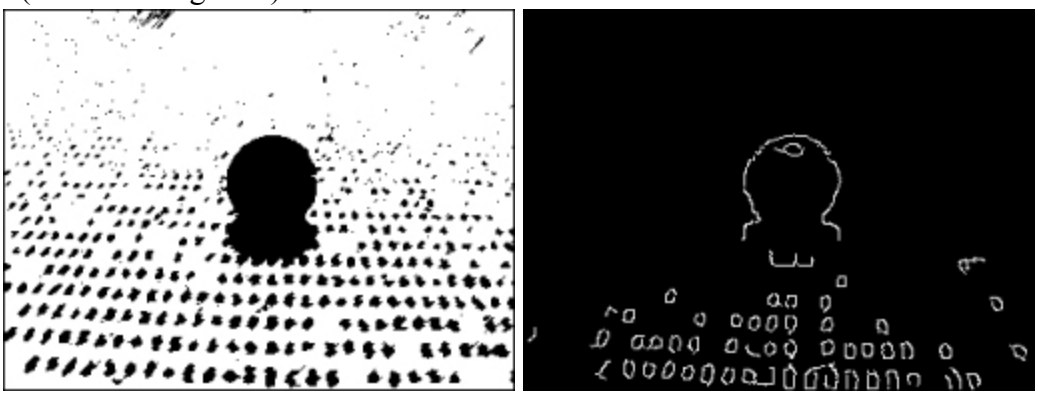


Figure 3 - Blob (left) and Edge (right) technique on non-uniform carpet

To resolve this there is a technique called Floor Finder Technique as seen in figure 4 (RoboRealm).

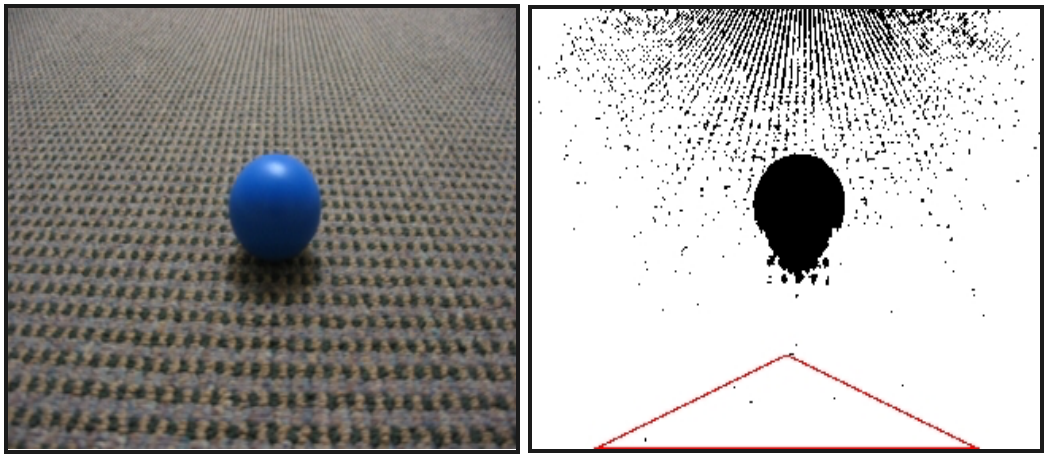


Figure 4 - Floor Finder on non-uniform carpet

The best technique for you will depend on the specific environment and what equipment is available.

Main point for this project is to take a picture after every inch/2 inches/foot a car would move forward. This way the car will get "the best path" within the image frame for the car to move without hitting objects. For this project, we did however have a few assumptions. Our

mobile robot will be on relatively flat ground and an indoor scene that a mobile robot may encounter while on the ground. Not only that, the carpet is a single color with the obstacles different colors than the ground plane. The camera on the car is placed at a downward angle of less than 75 degrees gives a field of view less than 4 feet for our laptop camera. To do this project, we 1st tested in Matlab, then moved to python.

When making the project, the following are the steps…

Step 1: Get the image.

In python, we took a bunch of real time pictures from the computer's built-in camera and tested our code with such images (as seen as Figure 5).

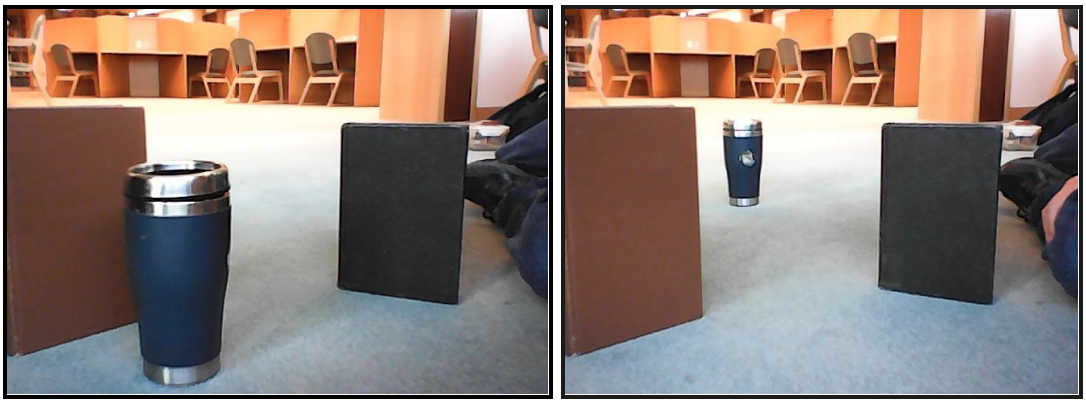


Figure 5 - Real Time images

Step 2: Turn RGB image to grayscale.

So we used the OpenCV built functions to turn the RGB images to grayscale.

Step 3: Calculate/Find edges.

In OpenCV, there are already built-in functions that search for the edges. We used those built-in OpenCV functions to find the edges. But for the built-in OpenCV edge detection we Used Canny Edge Detection method (as seen as Figure 6) to find edges within the image with a threshold of about 31-40.

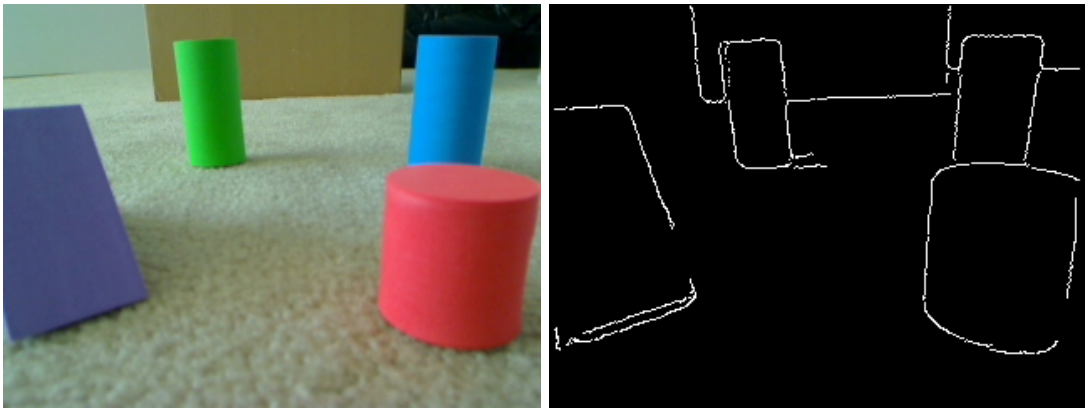


Figure 6 - Original image (left) & Canny Edge Detection (right)

Step 4: Fill the empty space.

With the edge-detected image, we took the size of the image. For this case, our size of the

image was 240 by 320 image. We went through the entire image starting from the bottom of the image in search of edges. Once we detected edges, then we fill from the edge to the top of the image with black or 0's.

Step 5: Open (erode then dilate) the filled image.

However because there was still a bunch of noise (carpet that was partly detected as edges) and thin lines in which a car might fit through. We had to open (erode and dilate) the filled the empty space.

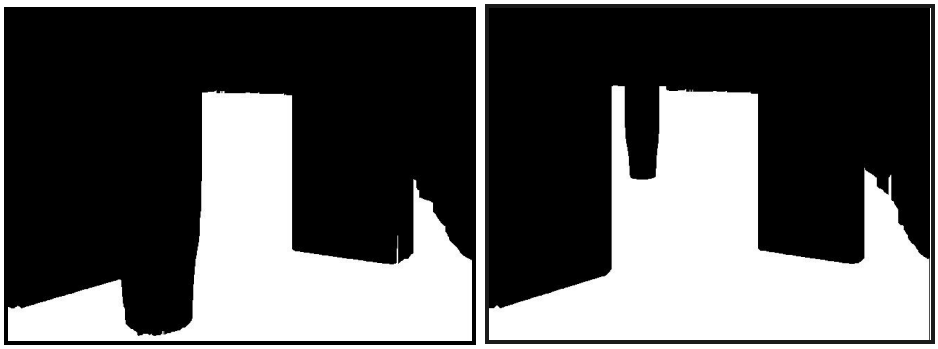


Figure 7 - Edge Detection Filled from the Bottom

Step 6: Combine the 2 images.

With the filled empty space and open images, I combine both images to visually see if it works and give the best point for the car to go.

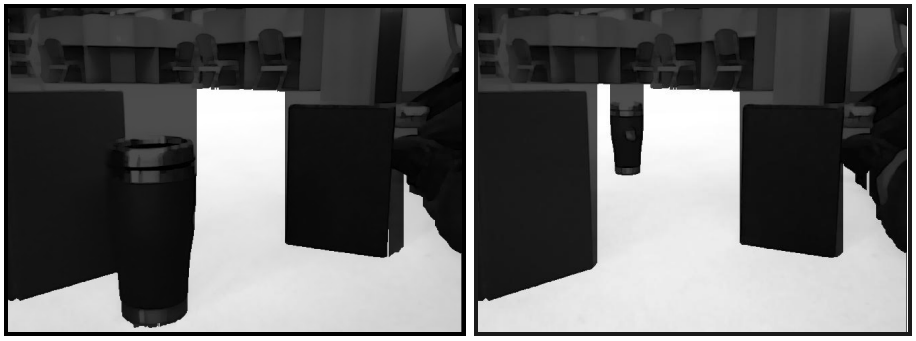


Figure 8 - Combined images

Step 7: Move the car a little & restart the process *(given if we had a car)*

However since we had no car, the car would essentially move forward in theory and the whole process would start all over. Starting with retaking an image, then detect edges.

Field of View

60, 70, 80, 90 degree field of views with a 6 foot measured range.

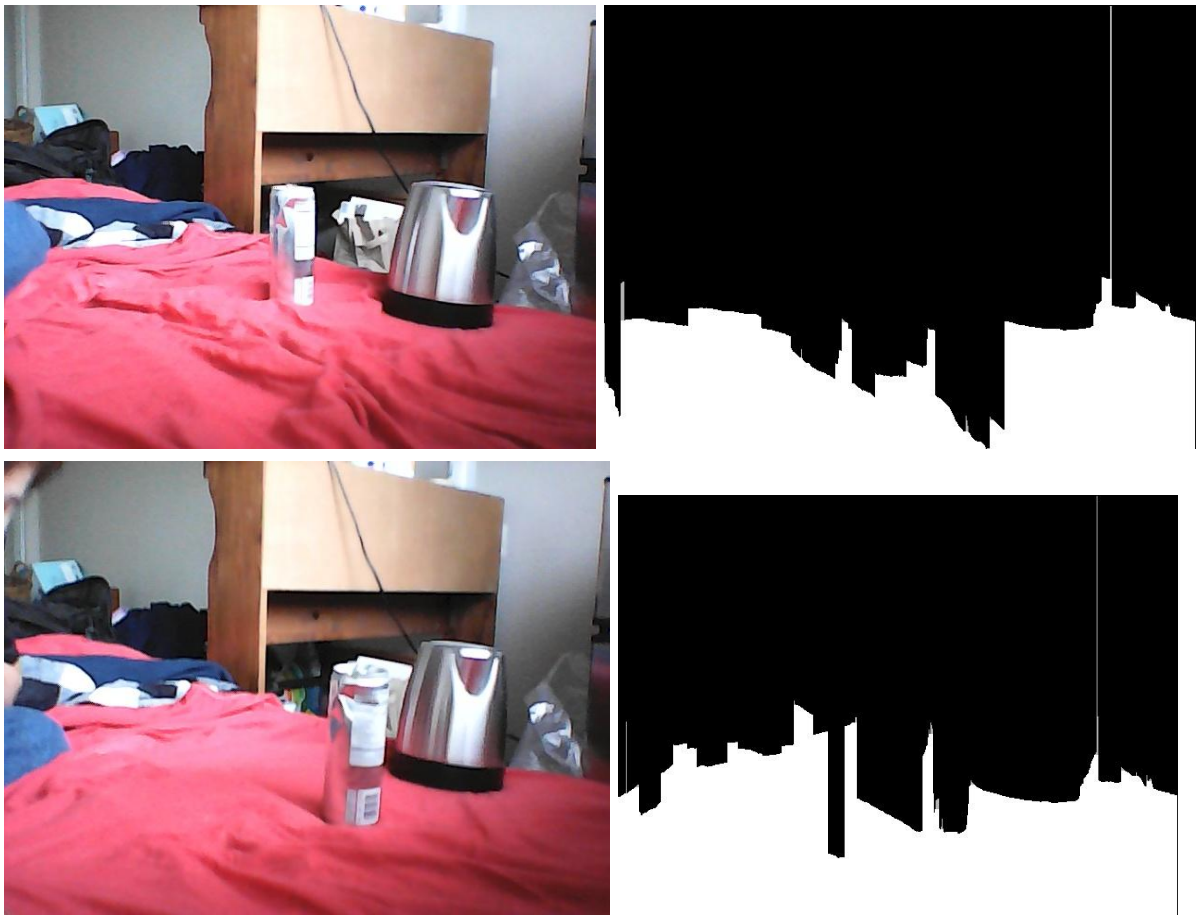
We experimented with optimal fields of view. Our method of creating paths relies on adequate vision of the ground immediately in front of us and a downward angle to be able to use the visual data of the ground to build a sense of depth. Contour lines of depth can be derived from either unbroken ground, a tape measure like we used, or range finding sensors. Future work on this project would derive more dimensions from the objects we're viewing with trig, identifying how large they are by their distance away, etc.

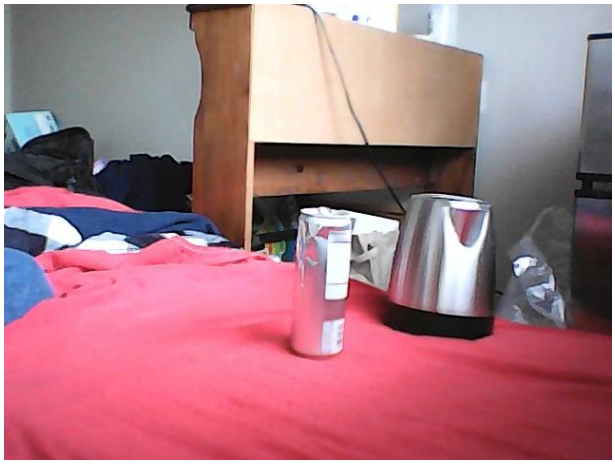
Runtime

Our algorithm obtained a frame periodically to run data on. The recorded time to complete that loop is 7 seconds.

Texture confusion

Our thresholding confused colors frequently, although was generally reliable for darker colors. Natural light is a serious problem, as the path was guided up the deodorant by its reflection.

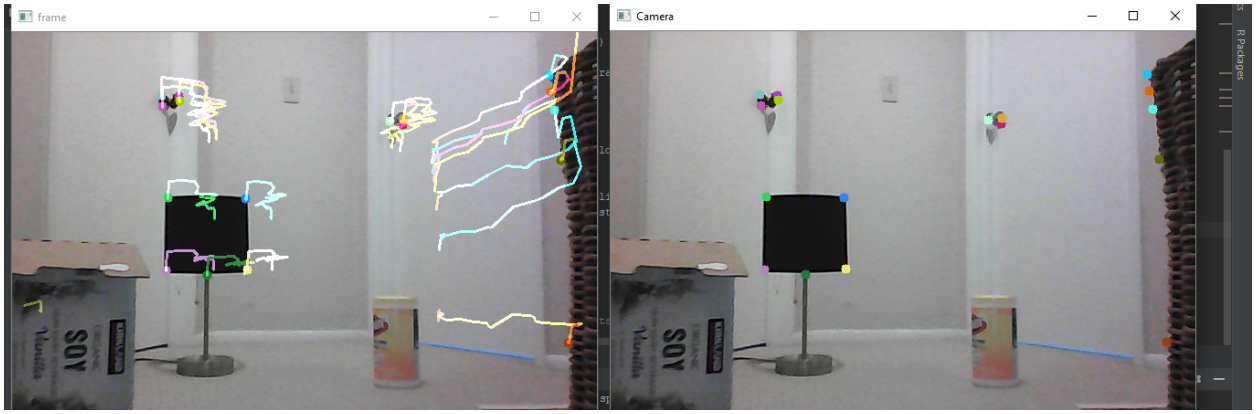


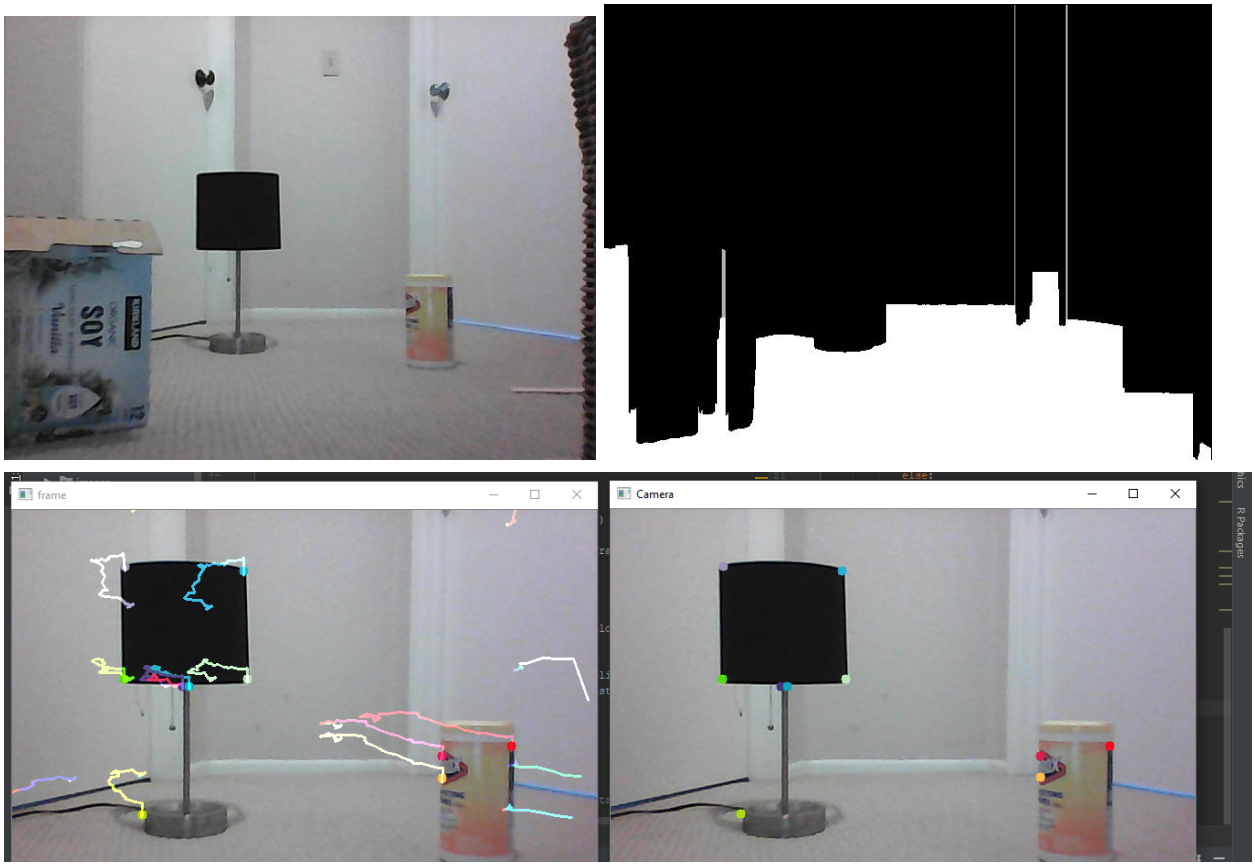


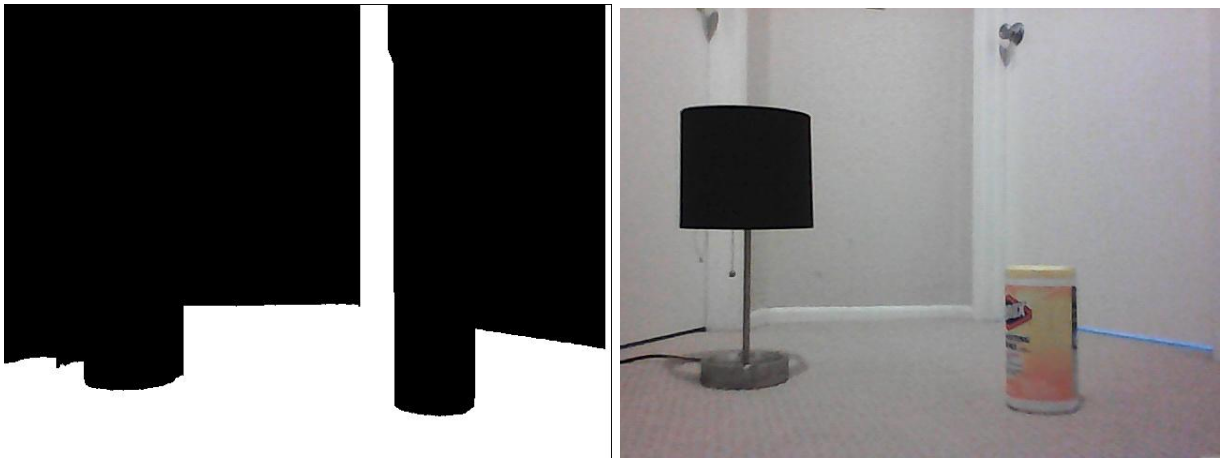

Here we observe the pathmaking interfered with by the texture of the bedsheet.

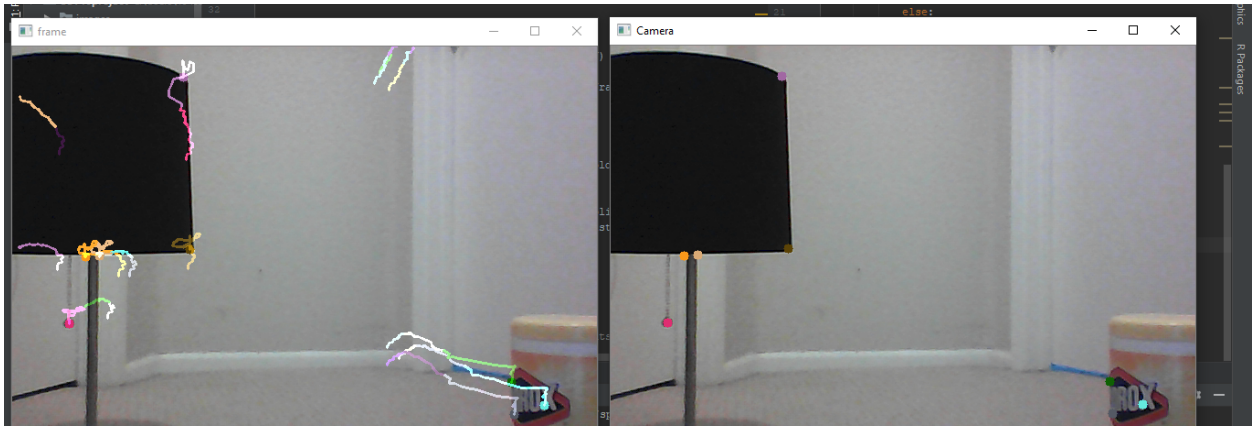
Optical flow testing






frame
Camera





Another area to explore is extracting optical flow data between updated frames. Our camera on the laptop wasn't stable. We could implement a raspberry pi and accelerometer to calculate position changes and improve stability. Utilizing optical flow would help us track objects above our fill, or as a way to detect errors in our thresholding of ground values. Additionally, we could continue to fill from other directions, which would take more exploration and research of the topics discussed in class.

CODE: sparceOpticalFlow.py

```
import numpy as np
import cv2 as cv
import cv2


#cap = cv.VideoCapture(args.image)
#cap = cv.VideoCapture('slow_traffic_small.mp4')
cap = cv.VideoCapture(0)

feature_params = dict( maxCorners = 100,
                       qualityLevel = 0.3,
                       minDistance = 7,
                       blockSize = 7 )

lk_params = dict( winSize  = (15,15),
                  maxLevel = 2,
                  criteria = (cv.TERM_CRITERIA_EPS | cv.TERM_CRITERIA_COUNT, 10,
0.03))

color = np.random.randint(0,255,(100,3))

ret, old_frame = cap.read()
old_gray = cv.cvtColor(old_frame, cv.COLOR_BGR2GRAY)
p0 = cv.goodFeaturesToTrack(old_gray, mask = None, **feature_params)

mask = np.zeros_like(old_frame)
stillFrameCount = 0
countLabel = 0

while(1):
```

```
    ret,frame = cap.read()
    frame_gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)

    p1, st, err = cv.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None,
**lk_params)

    good_new = p1[st==1]
    good_old = p0[st==1]

    for i,(new,old) in enumerate(zip(good_new, good_old)):
        a,b = new.ravel()
        c,d = old.ravel()
        mask = cv.line(mask, (a,b),(c,d), color[i].tolist(), 2)
        frame = cv.circle(frame,(a,b),5,color[i].tolist(),-1)

    img = cv.add(frame,mask)
    cv.imshow('frame',img)
    k = cv.waitKey(30) & 0xff
    if k == 27:
        break

    old_gray = frame_gray.copy()
    p0 = good_new.reshape(-1,1,2)
    cv.imshow('Camera', frame)
    if cv.waitKey(1) & 0xFF == ord('q'):
        break

    if stillFrameCount == 30:
        cv.imwrite('flowImages/mask{:d}.jpg'.format(countLabel), mask)
        countLabel+=1
        stillFrameCount = 0
        mask = np.zeros_like(old_frame)
cap.release()

cv.destroyAllWindows()
```

CODE: fillFromBottom.py

```
import cv2
import numpy as np
import imghdr

#pretty good! fillfrombottom and filltest

def fillImage(img):
    height = img.shape[0]
    width = img.shape[1]
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    img_blur = cv2.blur(gray, (3, 3))
    #cv2.imshow('name', gray)
    I_edge = cv2.Canny(img_blur, 38, 38*3, 3)
    #cv2.imshow('name1', I_edge)
    sizePath = [height, width]
    I_path = np.zeros(sizePath)
    #zeros are black, ones are white
    for i in range(height-1, 0, -1):
        for j in range(0, width-1):
            #print(I_edge[i,j])
            if I_edge[i, j] == 255:
                I_path[i, j] = 0
```

```
            else:
                I_path[i, j] = 255
    #cv2.imshow('intermediary', I_path)
    for i in range(height-1, 0, -1):
        for j in range(0, width-1):
            if I_path[i, j] == 0:
                for k in range(i-1, 0, -1):
                    I_path[k, j] = 0
    return I_path


cap = cv2.VideoCapture(0)
count = 0
countLabel = 0

kernel = np.ones((5,5),np.uint8)
while(1):
    ret, frame = cap.read()

    if ret:
        count +=1
        if count % 3 == 0:
            fill = fillImage(frame)
            opening = cv2.morphologyEx(fill, cv2.MORPH_OPEN, kernel)
            #grayOpening = cv2.cvtColor(opening, cv2.COLOR_BGR2GRAY)
            #grayIp = cv2.cvtColor(I_p, cv2.COLOR_BGR2GRAY)
            blend = cv2.addWeighted(fill, 0.7, opening, 0.3, 0, dtype = cv2.CV_32F)

            cv2.imwrite('images/{:d}_frame.jpg'.format(countLabel), frame)
            cv2.imwrite('images/{:d}_fill.jpg'.format(countLabel), fill)
            cv2.imwrite('images/{:d}_opening.jpg'.format(countLabel), opening)
            cv2.imwrite('images/{:d}_blend.jpg'.format(countLabel), blend)

            #count += 1  # i.e. at 30 fps, this advances one second
            cap.set(1, countLabel)
            countLabel += 1
    else:
        cap.release()
        break

    if count % 3 == 0:
        print(countLabel)
```

In conclusion, we found that lighting matters. Depending on the lighting, the different types of colors you will receive within the image (eg. with white, equal distribution of RGB… with yellow light, get more Green and Blue) results in messing up your edge detection technique and “best path” detection. The resulting edge based techniques will not work if you do not have the same colored carpet as edge detection will pick up on the small patterns of the carpet and incorrectly see them as obstacles. Edge Detection is not the most efficient in real time. It takes a lot of time to compute & will have to compute a new path for every image. To fix this a possible solution is to combine multiple methods to edge based methods. For future possible implementation, we would also move to a raspberry pi and supplement optical flow calculations with an accelerometer. Raspberry Pi has better and faster processing power than our personal laptops.