Rover Project Notebook by Raymond Andrade

This notebook contains the functions used in my variation of the Rover Project Challenge

~ Project Overview

The goal of this project is to take pictures taken from the front of a Rover, in this case a simulated Rover, and use perception techniques to identify points of navigable terrain, unnavigable terrain, and yellow colored rocks. Then a overhead map will be created with the data to map the positions of the terrain and rocks. The program must also use the data to make decisions on where to navigate while controlling the speed and direction of the Rover.

~ Data and Calibration

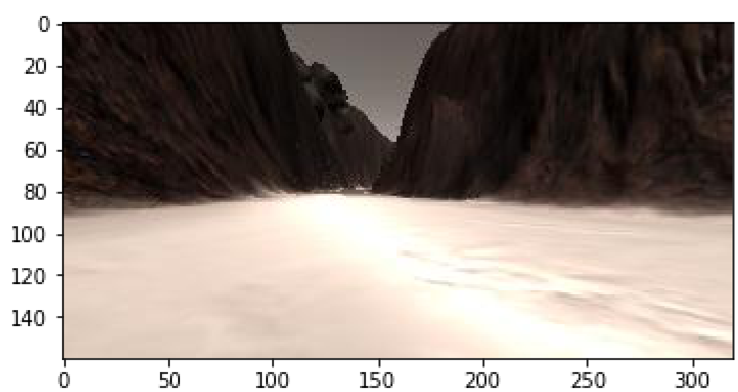
The following is the code for all imports needed throughout this project as well as a display of what the input data that will be process looks like. Run the code again to see another example image.

Note: You must be in RoboND environment

In [1]:

```
%matplotlib inline
#%matplotlib qt # Choose %matplotlib qt to plot to an interactive win
dow (note it may show up behind your browser)
# Make some of the relevant imports
import cv2 # OpenCV for perspective transform
import numpy as np
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import scipy.misc # For saving images as needed
import glob  # For reading in a list of images from a folder
import imageio
imageio.plugins.ffmpeg.download()

example_grid = '../calibration_images/example_grid1.jpg'
example_rock = '../calibration_images/example_rock1.jpg'
grid_img = mpimg.imread(example_grid)
rock_img = mpimg.imread(example_rock)

path = '../test_dataset/IMG/*'
img_list = glob.glob(path)
# Grab a random image and display it
idx = np.random.randint(0, len(img_list)-1)
image = mpimg.imread(img_list[idx])
plt.imshow(image)
```

Out[1]: <matplotlib.image.AxesImage at 0x1e92ed58048>



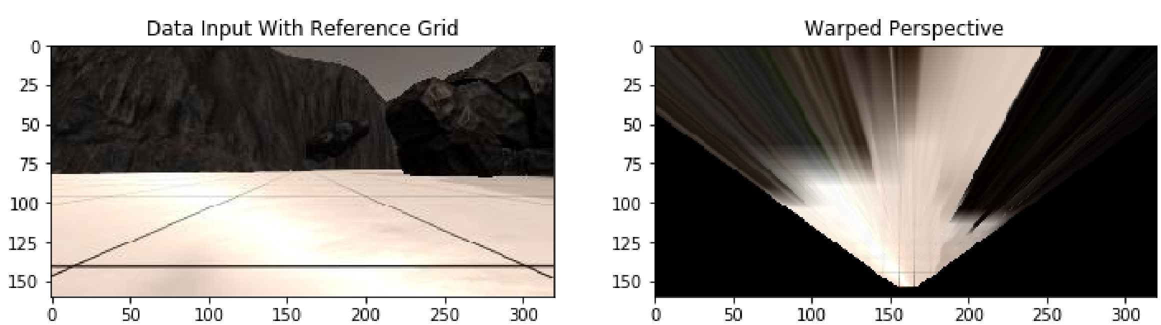
Transforming Perspective (Part 1)

Using numpy, the photo data can be rearranged to hep make an estimated overhead perspective based on what the Rover sees in front of it

In [2]:

```
def perspect_transform(img, src, dst):

    M = cv2.getPerspectiveTransform(src, dst)
    warped = cv2.warpPerspective(img, M, (img.shape[1], img.shape[0
]))# keep same size as input image
    mask = cv2.warpPerspective(np.ones_like(img[:,:,0]), M, (img.shap
e[1], img.shape[0]))

    return warped, mask

#set offset
dst_size = 5
bottom_offset = 6

#set transform range
source = np.float32([[14, 140], [301 ,140],[200, 96], [118, 96]])
destination = np.float32([[image.shape[1]/2 - dst_size, image.shape[0
] - bottom_offset],
                  [image.shape[1]/2 + dst_size, image.shape[0] - bott
om_offset],
                  [image.shape[1]/2 + dst_size, image.shape[0] - 2*ds
t_size - bottom_offset],
                  [image.shape[1]/2 - dst_size, image.shape[0] - 2*ds
t_size - bottom_offset],
                  ])

#Transorm
warped, mask = perspect_transform(grid_img, source, destination)

#Plot
fig = plt.figure(figsize=(12,3))
plt.subplot(121)
plt.title('Data Input With Reference Grid')
plt.imshow(grid_img)
plt.subplot(122)
plt.title('Warped Perspective')
plt.imshow(warped)
#scipy.misc.imsave('../output/warped_example.jpg', warped)
```

Out[2]: <matplotlib.image.AxesImage at 0x1e92f0968d0>



~ Transforming Perspective (Part 2)

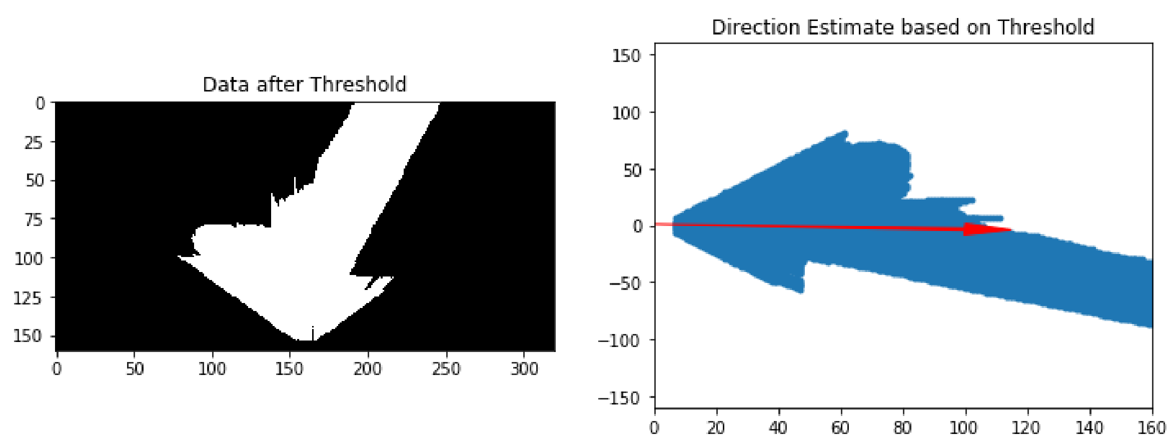
By reading the pictures in as a 3D array of pixel colors, in this case Red, Green, and Blue, it is possible to set a tollerance level for each to try distinguish the dark rocks from the light sand. Using the smae picture data from above, we can create a "desired path direction" indicated by the red line on the 2nd graph.

In [3]:

```
def color_thresh(img, rgb_thresh=(145, 145, 145)):
    # Create an array of zeros same xy size as img, but single channe
l
    color_select = np.zeros_like(img[:,:,0])
    # Require that each pixel be above all three threshold values in
 RGB
    # above_thresh will now contain a boolean array with "True"
    # where threshold was met
    above_thresh = (img[:,:,0] > rgb_thresh[0]) \
                & (img[:,:,1] > rgb_thresh[1]) \
                & (img[:,:,2] > rgb_thresh[2])
    # Index the array of zeros with the boolean array and set to 1
    color_select[above_thresh] = 1
    # Return the binary image
    return color_select

def rover_coords(binary_img):
    # Identify nonzero pixels
    ypos, xpos = binary_img.nonzero()
    # Calculate pixel positions with reference to the rover position
 being at the
    # center bottom of the image.
    x_pixel = -(ypos - binary_img.shape[0]).astype(np.float)
    y_pixel = -(xpos - binary_img.shape[1]/2 ).astype(np.float)
    return x_pixel, y_pixel


def to_polar_coords(x_pixel, y_pixel):
    # Convert (x_pixel, y_pixel) to (distance, angle)
    # in polar coordinates in rover space
    # Calculate distance to each pixel
    dist = np.sqrt(x_pixel**2 + y_pixel**2)
    # Calculate angle away from vertical for each pixel
    angles = np.arctan2(y_pixel, x_pixel)
    return dist, angles


def rotate_pix(xpix, ypix, yaw):
    # Convert yaw to radians
    yaw_rad = yaw * np.pi / 180
    xpix_rotated = (xpix * np.cos(yaw_rad)) - (ypix * np.sin(yaw_rad
))

    ypix_rotated = (xpix * np.sin(yaw_rad)) + (ypix * np.cos(yaw_rad
))
    # Return the result
    return xpix_rotated, ypix_rotated

def translate_pix(xpix_rot, ypix_rot, xpos, ypos, scale):
    # Apply a scaling and a translation
    xpix_translated = (xpix_rot / scale) + xpos
    ypix_translated = (ypix_rot / scale) + ypos
    # Return the result
    return xpix_translated, ypix_translated
```

```
def pix_to_world(xpix, ypix, xpos, ypos, yaw, world_size, scale):
    # Apply rotation
    xpix_rot, ypix_rot = rotate_pix(xpix, ypix, yaw)
    # Apply translation
    xpix_tran, ypix_tran = translate_pix(xpix_rot, ypix_rot, xpos, yp
os, scale)
    # Perform rotation, translation and clipping all at once
    x_pix_world = np.clip(np.int_(xpix_tran), 0, world_size - 1)
    y_pix_world = np.clip(np.int_(ypix_tran), 0, world_size - 1)
    # Return the result
    return x_pix_world, y_pix_world


warped, mask = perspect_transform(grid_img, source, destination)
threshed = color_thresh(warped)

# Calculate pixel values in rover-centric coords and distance/angle t
o all pixels
xpix, ypix = rover_coords(threshed)
dist, angles = to_polar_coords(xpix, ypix)
mean_dir = np.mean(angles)

# Plot
fig = plt.figure(figsize=(12,9))
plt.subplot(221)
plt.title('Data after Threshold')
plt.imshow(threshed, cmap='gray')
plt.subplot(222)
plt.title('Direction Estimate based on Threshold')
plt.plot(xpix, ypix, '.')
plt.ylim(-160, 160)
plt.xlim(0, 160)
arrow_length = 100
x_arrow = arrow_length * np.cos(mean_dir)
y_arrow = arrow_length * np.sin(mean_dir)
plt.arrow(0, 0, x_arrow, y_arrow, color='red', zorder=2, head_width=1
0, width=2)
```

```
Out[3]: <matplotlib.patches.FancyArrow at 0x1e92f14d278>
```

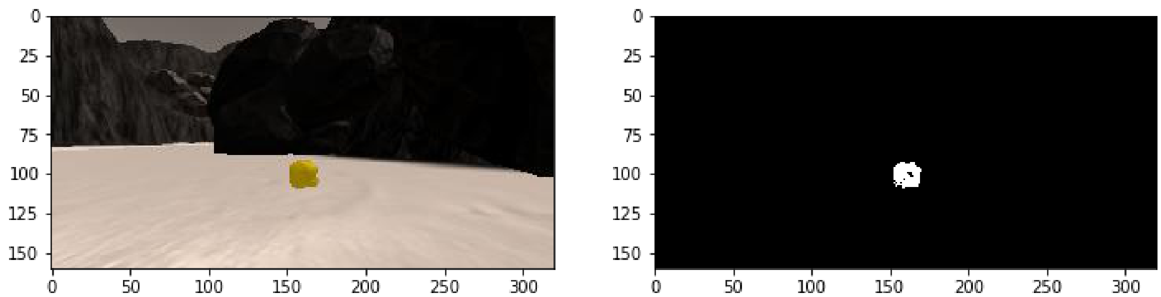


~ Identifying Rocks

Using the same concept as above, the picture is scanned for a particualr color threshold value, in this case yellow, to identify the yellow rocks.

In [4]:
```python
def find_rocks(img):
    rock_location = ((img[:,:,0] > 110) & (img[:,:,1] > 110) & (img[:,:,2] < 50))

    color_select = np.zeros_like(img[:,:, 0])
    color_select[rock_location] = 1
    return color_select

# Plot
fig = plt.figure(figsize=(12,9))
plt.subplot(223)
plt.imshow(rock_img)
plt.subplot(224)
plt.imshow(find_rocks(rock_img), cmap="gray")
```

Out[4]:
```
<matplotlib.image.AxesImage at 0x1e92f1d9470>
```



~ Processing Image Data

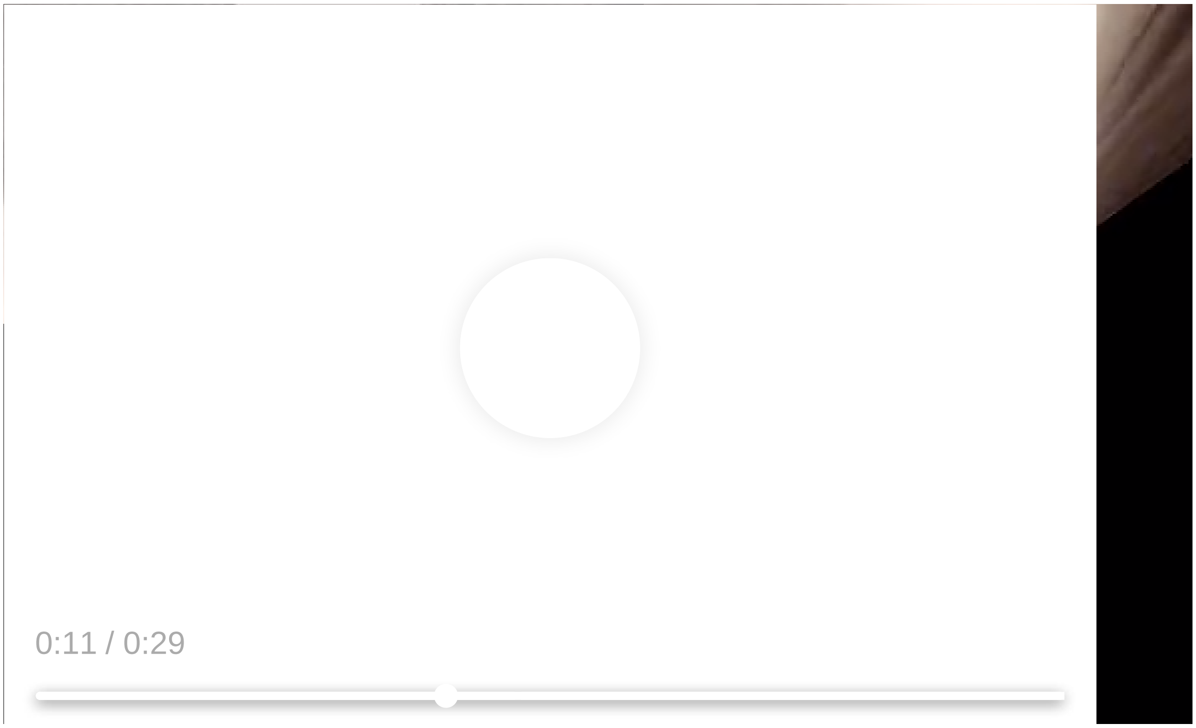
Below is a data bucket which will contain information from the CSV file to reference the sensor readings from the rover to the coresponding image data. The overhead map is created by layering different colors based on what is being identified in the camera.

In [5]:

```
# Import pandas and read in csv file as a dataframe
import pandas as pd

df = pd.read_csv('../test_dataset/robot_log.csv', delimiter=';', deci
mal='.')
csv_img_list = df["Path"].tolist() # Create list of image pathnames
# Read in ground truth map and create a 3-channel image with it
ground_truth = mpimg.imread('../calibration_images/map_bw.png')
ground_truth_3d = np.dstack((ground_truth*0, ground_truth*255, ground
_truth*0)).astype(np.float)

# Databucket to store info from CSV file
class Databucket():
    def __init__(self):
        self.images = csv_img_list
        self.xpos = df["X_Position"].values
        self.ypos = df["Y_Position"].values
        self.yaw = df["Yaw"].values
        self.count = 0 # This will be a running index
        self.worldmap = np.zeros((200, 200, 3)).astype(np.float)
        self.ground_truth = ground_truth_3d # Ground truth worldmap

data = Databucket()


def process_image(img):

    #mapping with color thresholding
    warped, mask = perspect_transform(img, source, destination)

    threshed = color_thresh(warped)

    map_o = np.absolute(np.float32(threshed) - 1) * mask

    x, y = rover_coords(threshed)

    world_size = data.worldmap.shape[0]

    scale = 2 * dst_size

    #retrieve position from bucket
    xpos = data.xpos[data.count]
    ypos = data.ypos[data.count]
    yaw = data.yaw[data.count]

    x_world, y_world = pix_to_world(x, y, xpos, ypos, yaw, world_size
, scale)

    #obstacle positioning
    ox , oy = rover_coords(map_o)
    ox_world , oy_world = pix_to_world(ox, oy, xpos, ypos, yaw, world
_size, scale)

    data.worldmap[y_world, x_world, 2] = 255
    data.worldmap[oy_world, ox_world, 0] = 255
    nav_pixel = data.worldmap[:,:,2] > 0
```

```
    data.worldmap[nav_pixel, 0] = 0

    map_r = find_rocks(warped)

    if map_r.any():
        rx, ry = rover_coords(map_r)
        rx_world, ry_world = pix_to_world(rx, ry, xpos, ypos, yaw, wo
rld_size, scale)
        data.worldmap[ry_world, rx_world, :] = 255




        # Create a blank image
    output_image = np.zeros((img.shape[0] + data.worldmap.shape[0], i
mg.shape[1]*2, 3))

        # Here I'm putting the original image in the upper left hand
 corner
    output_image[0:img.shape[0], 0:img.shape[1]] = img

        # Add the warped image in the upper right hand corner
    output_image[0:img.shape[0], img.shape[1]:] = warped

        # Overlay worldmap with ground truth map
    map_add = cv2.addWeighted(data.worldmap, 1, data.ground_truth, 0.
5, 0)
        # Flip map overlay so y-axis points upward and add to output_
image
    output_image[img.shape[0]:, 0:data.worldmap.shape[1]] = np.flipud
(map_add)


        # Then putting some text over the image
    cv2.putText(output_image,"Test Run in Training Mode ", (20, 20),
                cv2.FONT_HERSHEY_COMPLEX, 0.4, (255, 255, 255), 1)
    if data.count < len(data.images) - 1:
        data.count += 1 # Keep track of the index in the Databucket()

    return output_image


# watch video clips

output = '../output/test_mapping.mp4'

from IPython.display import HTML
import io
import base64
video = io.open(output, 'r+b').read()
encoded_video = base64.b64encode(video)
HTML(data='''<video alt="test" controls>
                <source src="data:video/mp4;base64,{0}" type="video/m
p4" />

</video>'''.format(encoded_video.decode('ascii')))
```

Out[5]:



Conclusion

When this code is applied to the Autonomous Rover Simulation, the Rover is able to maneuver around by itself while applying the map perspective.

On average, the Rover is generally able to map out 40% of the terrain in ~140 seconds with ~71% fidelity. It will also pick up yellow rocks that are directly in front of it as long as the Rover is not moving too fast.