Finding Lane Lines on the Road

Writeup Template

You can use this file as a template for your writeup if you want to submit it as a markdown file. But feel free to use some other method and submit a pdf if you prefer.

Finding Lane Lines on the Road

The goals / steps of this project are the following:

- Make a pipeline that finds lane lines on the road
- Reflect on your work in a written report

Reflection

1. Describe your pipeline. As part of the description, explain how you modified the draw_lines() function.

My pipeline consisted of 5 steps. These steps are shown below:

1. **Grayscale**

The Grayscale step takes an image and converts it to grayscale. Here, I was given a color image. A color image is represented by an array of pixels and each pixel possesses three values RGB (red, green, and blue) that constitute the output color. Below is the original color image and the grayscale converted image.

Figure 1: Color Image

Figure 2: Grayscale Image

With the help of OpenCV, these RGB values are converted to grayscale reducing the complexity of dealing with three values to one value. The OpenCV function used is called cvtColor(). This function can convert images to other representations like HSV (hue, saturation, and value). The function is shown as used below.

gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

2. Gaussian Smoothing

The Gaussian Smoothing step takes an image and blurs the image. This blurring effect reduces unwanted details and textures that may produce undesired results with the Canny Filter. To accomplish this, OpenCV has another function that can apply a Gaussian filter to an image. For the input, I take the grayscale image and define a kernel size which defines the size of the Gaussian filter. A larger kernel size applies more smoothing.

Figure 3: Grayscale Image

Figure 4: Gaussian Smoothing

The Gaussian blur is applied by dividing the process into two passes, one for the horizontal, and the other a vertical. The result is the same as convolving with a two-dimensional kernel but requires fewer calculations. The OpenCV Gaussian Blur function is shown below.

```
kernel_size = 11
blur_gray = cv2.GaussianBlur(gray,(kernel_size, kernel_size),0)
```

3. Canny Filter

The Canny Filter is an algorithm that takes an image and detects the edges. The Canny Filter is broken down into four steps.

1. Smooth the image using a Gaussian filter
2. Compute the gradient intensity representations of the image
    a. Filter with Sobel kernel in both horizontal and vertical direction (Gx, Gy)
    b. Calculate for edge gradient and direction for each pixel
3. Apply non-maximum suppression to remove "false" responses to edge detection
    a. Find local maximum in its neighborhood
    b. Suppress non-local maximum
4. Exercise minimum and maximum thresholds on the gradient values
    a. Values > max thresholds are kept
    b. Values < min thresholds are suppressed
    c. Values in between are kept based on connectivity to kept values

The resulting effect produces an image of mainly edges of the original as shown below.

Figure 5: Canny Filter

OpenCV provides a function for Canny Filters called Canny() that takes the source, a low threshold, and a high threshold. It is recommended that the low threshold and high threshold be values of 1:2 to 1:3. The OpenCV functions is shown below.

```
low_threshold = 50

high_threshold = 150

edges = cv2.Canny(blur_gray, low_threshold, high_threshold)
```

4. Region Masking

The Region Masking step removes unnecessary image information by filtering everything but the pixels that reside in a specified area. Here, the image has a field of view that includes the backdrop of mountains and areas outside of the road that are irrelevant to lane detection. The region mask filters this data and returns only the area specified.

For a car lane, a quadrilateral region mask filters the area of interest. To create the mask, the Canny Filtered image is converted to a zero array. Four points are defined in an array. Then a color value is selected. These three values are passed into the OpenCV fillPoly() function. This creates the mask that is bitwise AND to the Canny Filtered image with the OpenCV function bitwise_and(). The resulting image is the Canny Filtered image without the side and background edges leaving only the lane

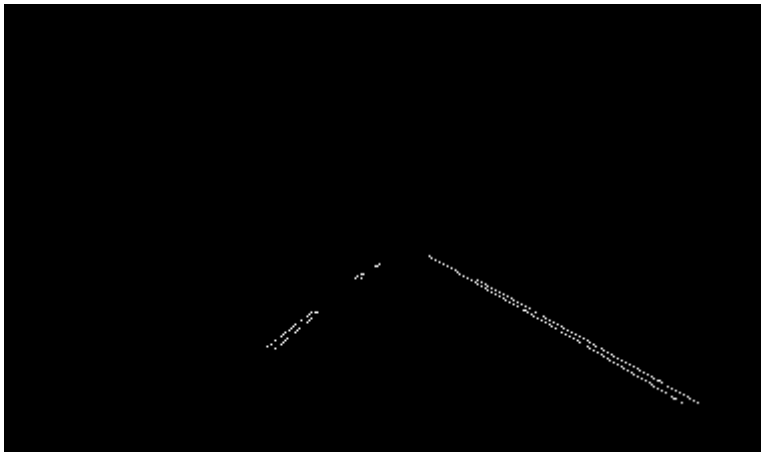

Figure 6: Region Mask Image

Example Code:

```
mask = np.zeros_like(edges)
ignore_mask_color = 255

imshape = image.shape
vertices = np.array([[(50,imshape[0]),(460, 330), (520, 330), (imshape[1] -
50,imshape[0])]], dtype=np.int32)
cv2.fillPoly(mask, vertices, ignore_mask_color)
masked_edges = cv2.bitwise_and(edges, mask)
```

5. Hough Filter

The Hough Filter detects a particular shape and in this case it is used to detect a line. The way that it accomplishes it is first by expressing the image space in polar coordinates changing cartesian coordinates (m, b) into polar coordinates (r, theta). For each point a sinusoid is created then those with intersections are incremented. If the number of intersections are above some threshold then it is deemed a line.

The OpenCV function takes multiple inputs first the image and an output array for the lines. The key configuration parameters are rho, theta, threshold, max_line_length, and max_line_gap. The rho provides the distance resolution of in pixels and the theta the angle resolution for the Hough grid. The threshold defines the number of intersections required. Then the min_line_length is the minimum number of pixels making up a line and the max_line_gap is the maximum gap in pixels between connectable line segments. Lastly, the output is an array containing the initial point and final point of the each line.

```
# Define the Hough transform parameters

    # Make a blank the same size as our image to draw on
```

```
rho = 2 # distance resolution in pixels of the Hough grid

theta = np.pi/180 # angular resolution in radians of the Hough grid

threshold = 15     # minimum number of votes (intersections in Hough grid cell)

min_line_length = 30 #minimum number of pixels making up a line

max_line_gap = 40    # maximum gap in pixels between connectable line segments

line_image = np.copy(image)*0 # creating a blank to draw lines on


# Run Hough on edge detected image

# Output "lines" is an array containing endpoints of detected line segments

lines = cv2.HoughLinesP(masked_edges, rho, theta, threshold, np.array([]),

                    min_line_length, max_line_gap)
```

6. Draw Lines

The Draw Lines function takes the lines and filters them to the lines that cover the lanes. The start of the algorithm loops through the line values. Within the loop, the algorithm takes a couple of steps:

1. Find Slope
2. Separate Left and Right Values
3. Generate Linear Fit (m b)
4. Extend end points to the full range of lane
5. Filter X and Y values outside of the image
6. Generate line in image

To find the slope I use a simple formula: m = (y2 - y1)/(x2-x1)

The left lanes are typically negative slope values and the positive slopes are right lanes. Furthermore, the lane slopes' values range are limited to values that fit lanes that are more vertical. The parameters are then separated into left and right lists.

The values are put into a polyfit() function to find a linear fit for all the right and left line values.

This function was used then to find the max and min values for the line based on the bottom of the image and the mask high y-value.

I then filtered X and Y min and max values outside of the visible image seeing those are false lines due to some noise or improper edge detection.

Lastly, the surviving lines are then passed to line() function to overlay an image.

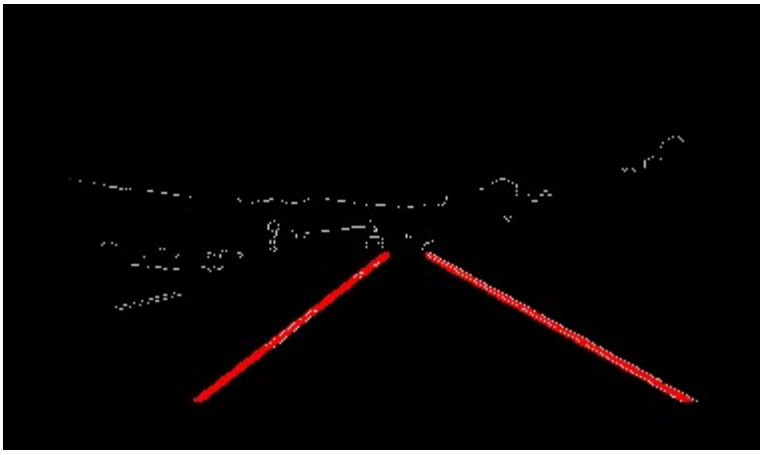

Figure 7: Draw Lines Function

2. Identify potential shortcomings with your current pipeline

One potential shortcoming would be what would happen when ...

Another shortcoming could be ...

I have a couple of shortcomings to this algorithm.

One is color saturation when the sun or lack of sun blurs the edge. This can lead to errors in the Canny Edge Detection. If it is significantly large then this can be a really challenging problem.

Another challenge would be handling turns at sharp angles. Angles are not straight lines so the polyfit() may need to be modified or changed to fit a curve.

The other challenge would be when things obstruct the lane lines. How do we build robustness to handle weather, vehicles, etc. that may cover the lane lines.

3. Suggest possible improvements to your pipeline

I was thinking of moving the mask earlier in the program to reduce the complexity of the calculations and increase speed. If only a portion of the image array needs to be processed then it be recombined at the end. I believe this would increase the speed performance and efficiency of this algorithm.

Another improvement would be to mask each lane instead of taking the middle section. This follows the same premise.

Another improvement would be to add some dynamic aspects to the pipeline. The program is currently static and it might not fit all possible scenarios. Dynamically changing this algorithm to best fit the situation based on other sensor data may produce a better outcome.