ENPH 353 - Fizz Detective Competition Report

April 21, 2024

Team 10 - Justin & Ryan

Table of Contents

Introduction

Background

The ENPH 353 Fizz detective competition is an opportunity to apply what we have learned throughout the course in an open and self-guided manner. We identified key concepts of the course to apply to our development of our agent: Computer vision applications, such as image processing and object tracking, ROS, software version control, and neural networks. For completing the course and scoring points, we decided that it was critical to have a smooth driving robot, so that our accuracy in detecting and submitting clues was high. We also thought it was important to not take any penalties, and so we wanted to complete the whole course without teleporting, or hitting any of the obstacles. After discussing, we decided to divide the work as follows: Justin took responsibility for the clue board detection, processing, and the convolutional neural network to read and return clues. Ryan took responsibility for driving and obstacle detection.

Software Architecture

Overall, driving was split into 4 regimes: Road, grass road, baby yoda, and mountain. By implementing different states for our robot, we were able to keep our driving control as clean and smooth as possible. We used the magenta lines as clear indicators of where to change states. As shown in Fig 1.1 below, this was adequately captured by 9 distinct states. The key differences between states was how the error was calculated. This involves different HSV masks and using different methods of evaluation, such as contours or iteration through image arrays.

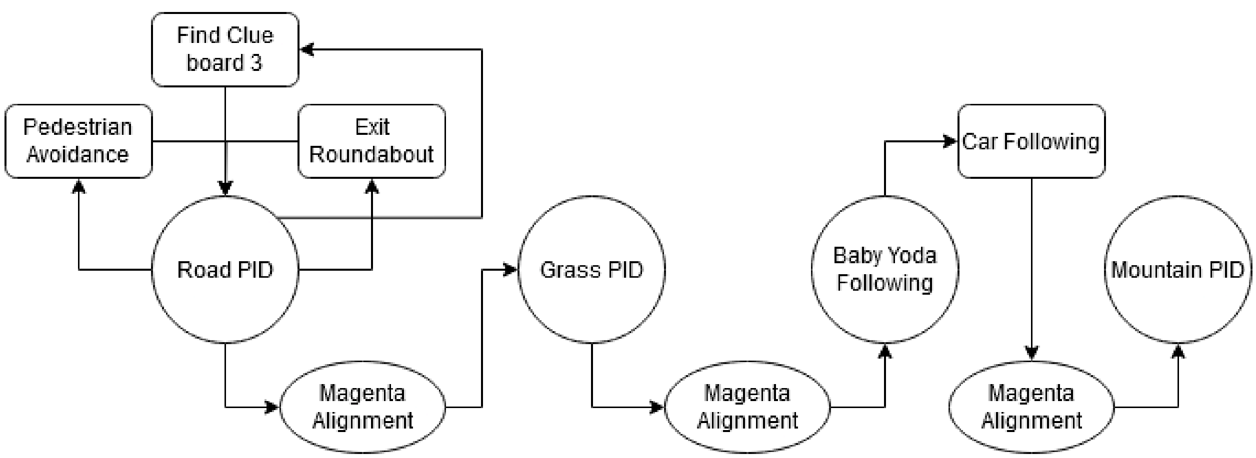


Fig 1.1 - Control State Diagram

With regards to clueboard recognition and prediction, the code was written in a fairly linear style with checks at various critical points that would determine whether the next major block of code would execute or not. In total, there were three critical checks behind which most of the image processing was locked. Two were dedicated to clueboard recognition and one was dedicated to clue prediction. The check for clue prediction was separate from clueboard recognition due to the way the code was designed but the only way to pass the check for clue prediction was to pass both checks for clueboard recognition. The clueboard recognition checks were nested checks so that it was possible to repeatedly execute the block of code unlocked by the first check until the second nested check could be passed.

Discussion

Robot Navigation

PID Navigation

Our initial plan was to use a convolutional neural network to drive our robot. However, once it was clear that a PID controller would be much more effective, we pivoted to that strategy. After discussing and testing with integral and derivative gain, we found that simply using proportional gain was sufficient. We believe that this is because we are working in a simulation, thus our robot has the ability to change its angular velocity nearly instantaneously.

Our PID controller was implemented using a weighted average of three centroids along the road, as displayed in Fig. 2.1. We decided to weight x3 the heaviest, as that gave us the best idea of where the road was heading, and x1 the least, as when the robot was centered on the roads, x1 was often in the center of the image.

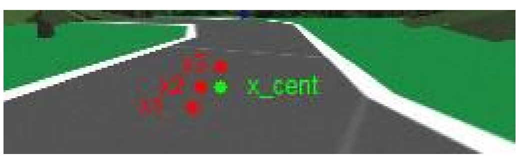


Fig. 2.1 - PID Image Processing

Fig. 2.2 - Gray Road Masking

To find the centroids, we downsized and cropped the image, converted the road to an HSV image, and masked the image for the gray road, as shown in Fig 2.2. By traversing across the three different heights and checking for white pixels, we summed the x values of white pixels and divided by the road width (number of white pixels). Finally, we took the weighted average of the three x centroids using the following formula: $x_{cent} = \frac{1}{6}(3x_3 + 2x_2 + x_1)$. This simple weighted average calculation allowed our robot to traverse the initial section of the environment smoothly, and well centered on the road, which was important for our clue board detection.

A peculiarity of our controller implementation was entering and exiting the roundabout. The robot entered the roundabout and traveled clockwise, which was desirable as the truck also travels clockwise, meaning we are less likely to run into it. However, we were unable to exit the roundabout only with a PID controller. Therefore, we implemented a condition to look for a tree near the exit, as shown in Fig 2.3. To find this specific tree, we utilized how reliable our controller was, and filtered for contours of above a minimum area with a y-centroid in a specific range. After finding this tree, we added a hardcoded turn to the left, and successfully found a consistent way to exit the roundabout.

Fig 2.3 - Exiting the Roundabout

Once our robot reaches the first magenta line, it must process the image it receives differently to properly implement PID. First, we check for magenta on the image by masking it in HSV, and we decide if we are close enough to it by checking a minimum area for a contour. Once we reach that minimum contour area, we stop traveling forward and focus on aligning ourselves with the magenta line. Using a rotated bounding box as shown in Fig 2.4, we set our error as the difference between 0 degrees and the angle of the bounding box, and rotate to correct it (2). Once the absolute error is less than 1 degree, we allow the robot to begin moving forwards again, this time changing to process the grass roads for control.

Fig 2.4 - Magenta Line Angle

The grass road is significantly more difficult to process than the gray road, due to the noise on the ground, light differences between the road lines and ground, and lack of difference between inside and outside the road lines. After some experimentation using the HSV masking script, we decided to first blur the image, then apply a grayscale, since the key differences between the road lines and the ground was the intensity, which is the value in HSV. Fig 2.5 shows a processed image before doing any centroid calculations. Instead of checking the road as in the previous section, we draw contours around the road lines, filtering for a minimum length. Then, at the same heights as before, we traversed across the image, noting where we intersected with a contour. We first check if we have intersections on both sides of the image. If we have intersections on one side, then we treat the road as filling the image up to that intersection. This is showcased in Fig. 2.6. This is so that the controller does not attempt to drive towards a single line if it happens to lose the other. Finally, to get the centroid of the road, we took the average of the two most extreme intersections. If there are no intersections, we treat the road as centered. The centroid is calculated in the same manner as the gray road.

Fig 2.5 - Masked Road

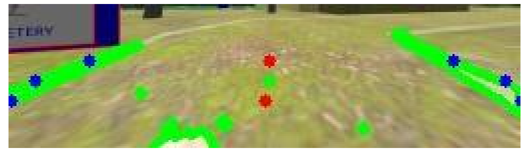


Fig 2.6 - Centroid Finding Method

Finally, our robot must climb up the mountain, where the road is similar to the forest road, but the sky and changes in lighting add additional difficulties. Our methods are the same, but involve a few new steps: First, before grayscaling, we mask out the sky so that no contours are drawn around it. Second, if we do not see any well defined contours, we change to an HSV filter, and use some of the noise to our advantage. Since the sides of the mountain are darker than the road, our image processing ignores the noise there, and instead our robot attempts to center itself based on the noise on the ground, until it can recapture good contours. Our backup method of finding centroids is shown in Fig 2.7.

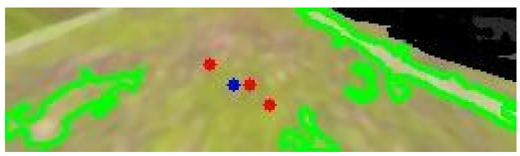


Fig. 2.7 - Backup Road Processing

In addition, our robot also slowed down when it encountered clue boards, in order to try and get clearer photos. We implemented a hard-coded turn for clueboard 3, since the angle of the board was not well suited to the path our controller took along that turn.

Obstacle Navigation

The first obstacle we encounter in the course is the pedestrian. We chose to detect the red line in a more simple fashion than the magenta lines. Since our PID control was consistent in the orientation, we simply checked if red pixels were in a range of y values, and stopped once that condition was met. We detected the pedestrian using an HSV mask for his blue jeans. At the crosswalk, the pedestrian was always the furthest down contour, so it was simple to pick out. An example of spotting the red line, and masking for the pedestrian are shown in Fig 2.8. We chose to wait for the pedestrian to make 1 cross before going. We accomplished this by waiting for the pedestrian to cross into the center of the image, then waited until it was on the side of the road before crossing.

Fig 2.8 - Pedestrian Masking

The second obstacle in the run is the truck in the roundabout. Fortunately, our speed was appropriate such that if the truck did not hit our robot at the intersection into the roundabout, we would avoid it altogether. To avoid this initial collision, we used another HSV mask, contours, and a minimum area to determine if the truck was on screen and close after we passed the third clueboard. An example of this masking can be seen in Fig 2.9. If our robot followed behind the truck, there was also a chance of rear-ending it. We kept the mask on and stopped our robot if it got too close to the back of the truck.

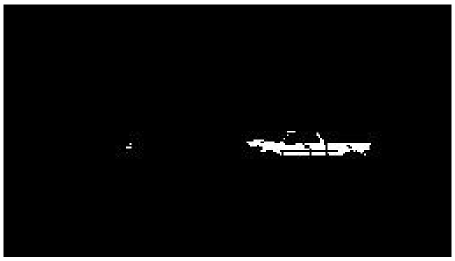


Fig 2.9a - Truck Masking

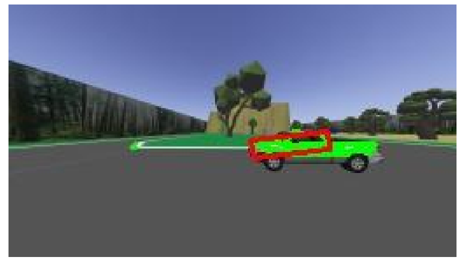


Fig 2.9b - Truck Contours

Finally, we used baby yoda to navigate to the base of the tunnel. We used an HSV mask to detect his cloak, as shown in Fig 2.10, and waited at the magenta line for him to come close enough before we began following him. We calculated our error as the difference between the centroid of his cloak, and the center of the image. In order to not collide with baby yoda, our linear speed x was a function of how close our robot was to baby yoda, measured by the lowest line given by the bounding box surrounding yoda's cloak, as shown in Fig 2.11. This function $x = x_o - x_o \frac{(y_{lower} - y_{far})}{(y_{max} - y_{far})}$ returns a speed, x_o when $y_{lower} = y_{far}$, and a speed of 0 when $y_{lower} = y_{max}$. Using this formula, we were able to ensure that our robot never runs into yoda. In order to get to the base of the tunnel, we masked for the windows of the car beside the tunnel, and changed from following yoda to following the car when the area of the contour became big enough. An example of this can be seen in Fig 2.11. Once we reach another area threshold, our robot stops moving, and rotates until it finds the magenta line, and aligns itself before going up the mountain.

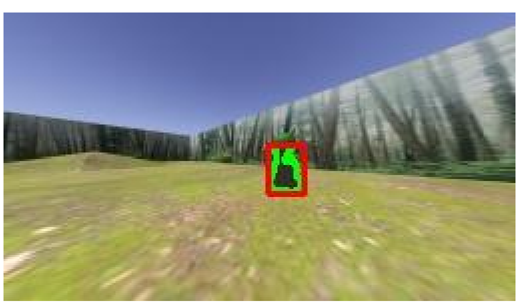


Fig 2.10 - Yoda with Bounding Box

Fig 2.11 - Truck Window Contours

Clue Board Recognition and Prediction

Working off previous experience in processing images with alphanumeric characters and in training convolutional neural networks, clue board recognition and prediction was accomplished with the use of Python's OpenCV library and Keras library. The full code for this module can be found in plate_predict_final.py in the competition Github repository.

Clueboard Detection

Clueboards in the competition simulation world are gray areas that contain dark blue text representing the clue. These gray areas are part of a larger board model that is the same shade of dark blue as the clue text. To detect the clueboard, we masked the camera feed image in HSV, setting a lower and upper threshold, to isolate for the dark blue of the clueboards.

Through this masking, we were able to use OpenCV's contour detection function, cv2.findContours(), to find the contours of the clueboards. Furthermore, the contours were approximated to polygonal curves using the cv2.approxPolyDP function. This function uses the Douglas-Pecker algorithm to reduce the number of vertices in a given contour. By approximating the contour to polygonal curves, we can now take advantage of the fact that we know that the clueboards are rectangles and thus have 4 vertices, 1 for each corner. By filtering all detected contours within the camera feed image by the number of points in each contour, we can find all rectangular contours and append them to a list.

Imposing the condition that the list has to have at least two elements in it and sorting the list by contour area in descending order, we can consistently identify the contour representing the gray area on the clueboard, which should be the second largest contour on the screen.

Fig 2.12 - Identifying the clueboard (contour in light green)

Once this contour is identified, it will be checked against a predetermined area threshold to ensure that the clueboard in the camera feed image is large enough to be upsized into a perspective-corrected image of good quality (i.e. not blurry; readable). If the contour passes the check, a perspective transform is performed to produce a flat image of the clueboard that is ideal to use for processing the characters on it.

For each clueboard in the simulation world, multiple transformed images may be produced for a single clueboard. To identify which transformed images belong to the same clueboard, there is a variable initialized within the code called frame_count. This variable increments every camera frame update and will be reset to 0 every time a transformed image is produced and appended to a list. At this point, the code will reach a block that can only be executed if the list holding the transformed images is not empty and the frame_count variable is greater than a certain value. Through this method, a "waiting period" of several frames must be elapsed without any transformed images being produced. This allows for continuity between consecutive frames of the same clueboard where one frame passed the conditions to be perspective transformed and the other did not. If the “waiting period” has elapsed, then we can be quite sure that the clueboard has completely passed out of the camera frame and the transformed images for the clueboard should be processed immediately in preparation for the next clueboard.

Each of the transformed images in the list are assigned a blur value, which is a float value equal to the variance of the transformed image’s Laplacian. The transformed image with the highest blur value is the least blurry image and this clue image will be selected to undergo further processing for clue prediction.

Clueboard Processing

The clue image is then cut into sections. The bottom section contains the clue value we want to predict. Using the same HSV thresholding parameters to find the clueboard, we can isolate the letters in the bottom half of the clue image and use cv2.findContours() to draw contours around each character. Dilating the contours using cv2.dilate() ensures that each character has no problematic gaps in its contour and joins the contours of letters in close proximity to each other.

Once dilated, a bounding box can be formed around each contour using cv2.boundingRect(). Any separate bounding boxes are then joined together to create a single bounding box that encompasses the entire clue. The return values from cv2.boundingRect() then allow us to crop the clue image according to the bounding box.

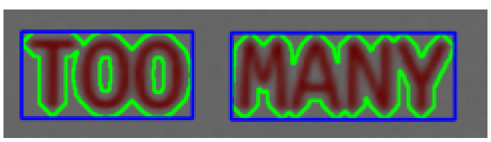


Fig 2.13 - Contours and bounding boxes

Fig 2.14 - Word crop

Before the clue crop is separated into individual character crops, its intensity must be normalized through modification of its HSV values, specifically the V values. The mean V value of the clue crop is calculated and if it is not within a certain range, the V value for each pixel is either incremented or decremented in code until the desired mean value is reached. By normalizing the clue crop’s V values, we are effectively modifying its intensity spectrum which directly translates to what it will look like in grayscale.

From this cropped image, multiple crops of constant width are taken across the image to isolate for each character. These crops are then grayscale and binarized using a binary threshold to produce binary images of the characters.

Fig 2.15 - Character crops

These binary images can then be fed into the neural network to predict what characters are in them. While spaces were originally included in the prediction capabilities of the network, the network was noticeably less consistent in correctly predicting spaces than any other character. As such, an alternative method to determine whether a character crop contained a space was used. Since the character crop for a space is made up of mostly 0 valued pixels, the mean pixel value of a space character crop is much lower than any other character crop. By thresholding the mean pixel value of each character crop, we identified spaces much more consistently than if we were to use the neural network.

Neural Network Description

The architecture of the neural network we developed is shown below in Fig 2.16. This network was trained entirely on images taken within the simulation. These images were gathered and labeled by a script. They were then processed into individual character crops as described in Clueboard Detection and Clueboard Processing. The network was trained for 20 epochs at a learning rate of 1×10^{-5} on the data set of characters in Fig 2.17.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 88, 43, 32)        320

 max_pooling2d (MaxPooling2  (None, 44, 21, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 42, 19, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 21, 9, 64)         0
 g2D)

 flatten (Flatten)           (None, 12096)             0

 dropout (Dropout)           (None, 12096)             0

 dense (Dense)               (None, 512)               6193664

 dense_1 (Dense)             (None, 37)                18981

=================================================================
Total params: 6231461 (23.77 MB)
Trainable params: 6231461 (23.77 MB)
Non-trainable params: 0 (0.00 Byte)
```

Fig 2.16 - Architecture

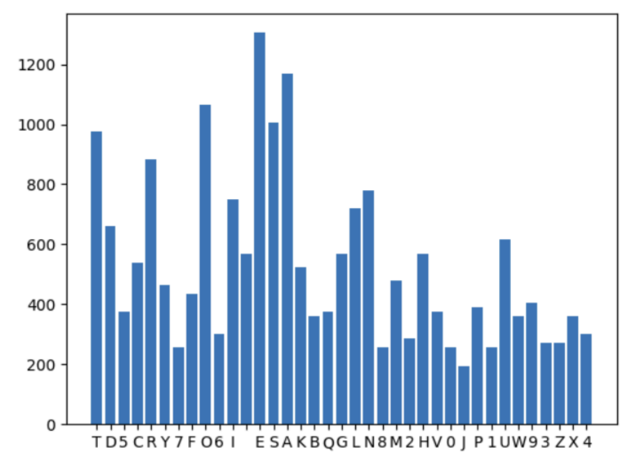
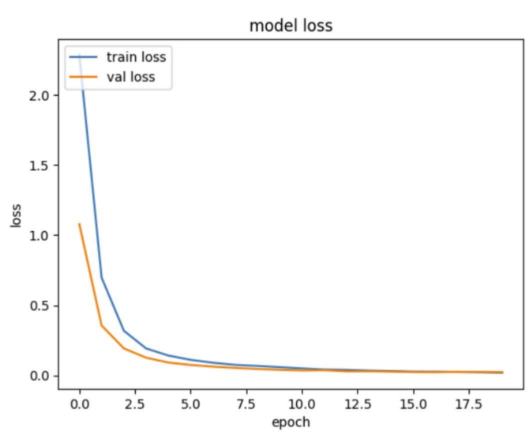
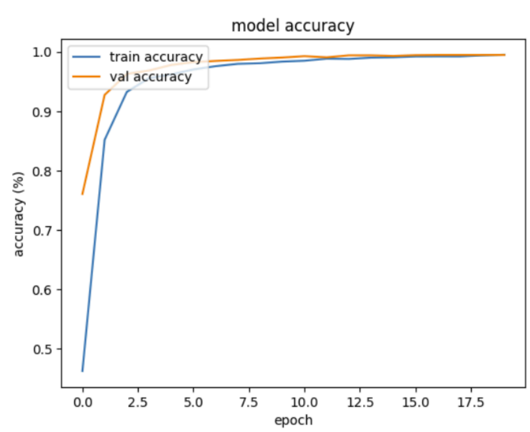


Fig 2.17 - Data set histogram

The model achieved a training loss of 0.0188, training accuracy of 0.9941, validation loss of 0.0187, and validation accuracy of 0.9952. During training, model loss and accuracy varied as shown below:





Conclusion

Competition Performance

During the competition, our robot unfortunately failed a turn and was unable to recover from it. Our robot managed to do the road part of the course perfectly, avoiding the pedestrian and truck while predicting all clues along the way correctly. Our robot took too sharp of a left turn on the transition to the grass part of the course and was unable to recover using PID, going offroad and getting stuck on the side of the mountain. At this point, we had collected 24 points and were deducted 2 points for going offroad. Our final score was thus set at 22 points. Our completion time was set to the maximum of 4 minutes since we had no way of stopping the timer other than by manually sending the stop message to the score tracker. We demonstrated that our agent was capable of making it through the rest of the course by resetting him to a spawn point in front of Baby Yoda where the robot was able to successfully complete the course.

Reflection

With regards to the clue recognition and prediction, we feel that the neural network performed very well. The image processing that went into training the network was, in our opinion, could have been improved in some areas. Ensuring that the clue word was straight before it was separated into characters was a concern during development and multiple efforts were made to do so. We attempted to straighten the clue word at various stages in our image processing.

One method was to add or strengthen the conditional checks on the camera feed image before taking a perspective transform of the clueboard. One such check we tested was an extent check. The extent of a contour is the ratio between its area and its bounding box area. This bounding box is a perfect rectangle with no rotation that encapsulates the entire contour so the closer the extent is to 1, the more perfectly rectangular the contour is. This would have helped straighten the image but it was not adopted because while testing the driving, we found that the way our robot drives often caused the clueboards to fail the extent check and so the robot would just skip clueboards very often. The extent check also did not seem to be as effective in straightening as we thought since at times, the contour approximation would cause some of the blue frame around the clueboard's gray area to be included in the perspective transform which ended up tilting the image regardless.

Another method was to try and use rotated bounding boxes on the clue images during the identification of the characters. This would have eliminated any reliance on the above method and would have actually been the most reliable method to straighten words before cropping into characters. The rotated bounding boxes would have aligned themselves to the tilt of the world and a following perspective transform would have produced an image of the straightened word. However, this technique was also not adopted due to difficulties in using OpenCV's function, cv2.minAreaRect(). Other than the inconsistency of function's return values, the biggest issue was the fact that some clue characters would have isolated contours even after dilation and due to the characters geometry, their rotated bounding boxes would not actually be aligned with the top and bottom of the character. Joining the bounding boxes to create one large box would then fail as can be seen in the appendix in Fig C1.

After much testing, we decided to abandon efforts to straighten the clue word since it would have proved to be a minor improvement to our image processing and our processing scheme was already functioning well without it.

In terms of what could have been done differently, we think that there could have been a better way to isolate the characters than with a hard coded width crop. While the width crop gave us good results, it would still have been ideal to isolate for each character in a different way. The width crop was sensitive to the orientation of the word and would often include some pixels of one character into the image crop of the adjacent character. In some edge cases, the error accumulated enough to cause concern over whether the data point was still a good data point or not. An example can be seen in Fig C2.

Instead of a hard coded width crop, we could have spent more time in pursuing alternative techniques like not dilating the character contours and using a bounding box for each separate character to determine the appropriate character crop. This was actually the first technique we attempted to use to isolate the characters but due to imperfections within the clue image caused by perspective transformation, the contours would often be difficult to process consistently. Issues included imperfect HSV thresholding resulting in anomalous contours surrounding random spots on the image, multiple contours being found for a single character, etc. While these issues could have been resolved, we felt there were simply too many issues and too many edge cases to resolve on our timeline. Given more time, we would have likely been able to develop this technique further and use it without concerns of inconsistency.

For our driving controller, the PID was incredibly consistent outside of a few areas. The controller had a significant bug when transitioning from the road to the grass road, where it would occasionally turn too hard to the left and fail to recapture the road, which is what occurred during our competition run. If given more time, we would have worked this bug out. We believe it was caused by the controller detecting the state change too early, and judging that the first few frames of grass road it could see were noisy on the left side. This caused it to bias left, missing which eventually led to the robot defaulting to going straight and missing the road altogether.

Other than bugs, we would have liked to see how fast our agent could have gone, while still managing to take good pictures of clue boards and staying consistent on the mountain. This would have required a combination of techniques above, and most likely smoother image processing for the grass roads.

References

1. https://pyimagesearch.com/2015/09/07/blur-detection-with-opencv/
2. https://namkeenman.wordpress.com/2015/12/18/open-cv-determine-angle-of-rotatedrect-minarearect/

Appendix

ChatGPT Logs

ChatGPT was minimally used in the design process. Our utilization of ChatGPT lay heavily in error debugging and in OpenCV documentation search. Most complicated code sourced by ChatGPT was incorrect or required heavy modification.

JU **You**

cv2.error: OpenCV(4.6.0) /io/opencv/modules/imgproc/src/imgwarp.cpp:3392: error: (-215:Assertion failed) src.checkVector(2, CV_32F) == 4 && dst.checkVector(2, CV_32F) == 4 in function 'getPerspectiveTransform'

ChatGPT

The error you're encountering typically arises when using `cv2.getPerspectiveTransform()` with source and destination points that don't meet the requirements. The error message suggests that the input source and destination points should be 4 points each with 2 dimensions (2D points).

To resolve this error, ensure the following:

1. **Number of Points:** Both source and destination points should have exactly 4 points each.
2. **Data Type:** The points should be of type `np.float32`.

JU **You**

cv2.error: OpenCV(4.6.0) :-1: error: (-5:Bad argument) in function 'warpPerspective'
> Overload resolution failed:
> - Can't parse 'dsize'. Sequence item with index 0 has a wrong type
> - Can't parse 'dsize'. Sequence item with index 0 has a wrong type

ChatGPT

The error message indicates that there's an issue with the format of the `dsize` argument passed to the `cv2.warpPerspective()` function. The `dsize` argument should be a tuple containing the width and height of the output image.

You

How can I only get contours of a certain length

ChatGPT

To filter out contours based on their length (perimeter), you can iterate over all the contours found and calculate the length of each contour using `cv2.arcLength()`. Then, you can decide whether to keep or discard each contour based on your length criteria. Here's how you can modify the previous code to achieve this:

python   Copy code

```python
import cv2

# Read the grayscale image
gray_image = cv2.imread('input_image.jpg', cv2.IMREAD_GRAYSCALE)

# Find contours
contours, _ = cv2.findContours(gray_image, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE

# Define the minimum and maximum length of contours you want to keep
min_contour_length = 50
max_contour_length = 200

# Draw contours on a blank image with the specified length range
contour_image = cv2.cvtColor(gray_image, cv2.COLOR_GRAY2BGR)
for contour in contours:
    contour_length = cv2.arcLength(contour, closed=True)
    if min_contour_length <= contour_length <= max_contour_length:
        cv2.drawContours(contour_image, [contour], -1, (0, 255, 0), 2)  # Draw the co

# Display the image with contours
cv2.imshow('Contours', contour_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

fix this error: error: OpenCV(4.8.0) :-1: error: (-5:Bad argument) in function 'circle'
> Overload resolution failed:
> - Can't parse 'center'. Sequence item with index 0 has a wrong type
> - Can't parse 'center'. Sequence item with index 0 has a wrong type

ChatGPT

The error you're encountering is due to the fact that `x_cent` is a NumPy array containing the x-coordinate of the center of the circle. However, `cv2.circle()` expects the center to be provided as a tuple of integers `(x, y)`.

To fix this error, you need to extract the value from the NumPy array and convert it to an integer. Here's the corrected code:

Data and Figures

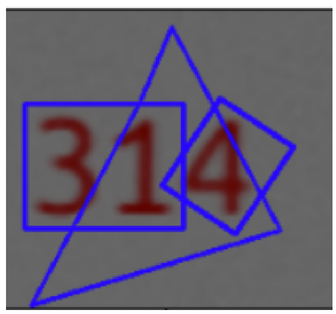


Fig C1 - Trying to use rotated bounding boxes on the characters

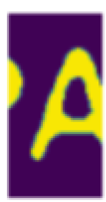


Fig C2 - Bad crop of ‘A’