CS585 Final Project: Predicting Steering Wheel Orientation for Application in Autonomous Vehicles

Muhammad Aseef Imran
Munir Siddiqui
Timothy Evdokimov

Table of Contents

Problem Definition

Our overall goal is to make a sort of scale model of a self-driving car. Given a first-person video of someone driving an automobile, we aim to segment the driver's view to isolate the dashboard, steering wheel, and road. This is in order to collect data on the curvature of the road and angle of the steering wheel, which we in turn use to build a model predicting the angle of the steering wheel based on road curvature.

Motivation

The development of an autonomous vehicle has been an extremely popular problem in industry for some years now, and is a problem heavily connected to computer vision. We wanted to get a feel for applying the tools we learned in CS585 to this problem, by attempting to solve for ourselves some of the segmentation and modeling problems in a purely simulated environment, using pre-recorded footage of people driving vehicles to both train and evaluate our models.

Much of this project was simply extracting the necessary data from the videos we used, which was motivated by a lack of good readily available datasets for autonomous driving. There are better ways to record the angle a car steers than doing computer vision analysis on a video of the steering wheel. However, such data was not readily available, so we wanted to use a method that would allow people to easily collect more autonomous driving data with little more than a phone camera and a car.

Assumptions

Our core assumptions are that our input data would have consistent lighting, an approximately monochromatic road and that the vehicle dashboard templates and position of the camera would be given.

Consistent lighting is required in order for our segmentation algorithm to work properly, since both our template matching and road segmentation systems are dependent on the pixels in the footage approximately matching our templates and expected road conditions. Similarly, our road detection algorithm depends on the road being approximately grey. We also expect the camera position to be from the driver's perspective in order for the template matching system to be able to adequately find key elements on the dashboard and steering wheel, and we expect the vehicle itself to be close enough to the vehicle in the training data to match our templates correctly.

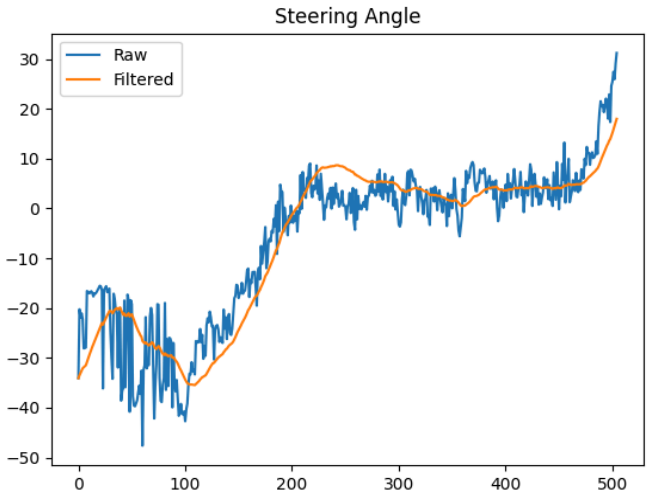
It is worth noting that these assumptions stemmed for our data; the training data we used was largely pulled from a car review magazine, which filmed first-person driving videos. These videos were filmed with a known car, from an approximately constant camera position and on a known road at a constant time of day, therefore inherently meeting our assumptions of camera position, lighting, road conditions, and vehicle.

Applications

The application in mind for this project is for an optical approach to a self-driving car. The idea being to use only data harvested from a single camera in the driver's first-person view to predict how to steer the vehicle. This is obviously sorely insufficient for a truly autonomous vehicle, but the segmentation approaches applied here could be transferred to a more complete self-driving car application. The approach of predicting the future state of a system using some set of past samples is also hardly novel, but nonetheless incredibly useful in the fields of control systems and autonomous vehicles. In short, our project was specifically designed to be a kind of "scale model" of a self-driving vehicle, implementing a small subset of technologies needed to drive a car autonomously is a purely simulated environment.

Challenges

There were a few major challenges encountered while developing the segmentation and modeling system, some intrinsic to the data and some intrinsic to the problem. For instance, a significant issue was caused by the data collected purely by segmentation of a hand-recorded video of a moving car proved very noisy. We were able to resolve this issue by applying an alpha-beta filter, which reduced noise significantly



The model itself proved quite challenging because the driver turns the wheel in reaction to the road *ahead* of them. Curvature in the road does not immediately affect steering angle, but rather a few frames later (on average, around 40 frames for our data at 30 fps).

Notice the "phase shift" between the two curves in the graph above. The resolution to this issue is to predict the next steering wheel angle based on the past N=100 road angle samples because the phase shift isn't constant but rather proportional to the vehicle speed. Feeding the past 100 road curvature measurements of any given point allows the model to account for the inconsistency in the phase shift and make a reasonably good prediction, given that we do not account for the vehicle speed in our current model.

Method and Implementation

Data

Our data collection process involved pulling first-person driving videos off of YouTube and then cropping them to include only relevant segments and parsing them frame-by-frame in OpenCV. The videos were sourced from a magazine that does car reviews, which proved very suitable since they all recorded their drive from a consistent camera angle and under approximately constant lighting conditions. The data totaled approximately 7 minutes of usable footage. The footage from the Audi test drive was primarily used, and most of our experiments were conducted on the 12000 frames from the file "audi_road_cropped.mp4."

The raw data is available in this Google Drive folder: 🗀 CS585_Project_Data

Algorithmic Steps and Implementation Details

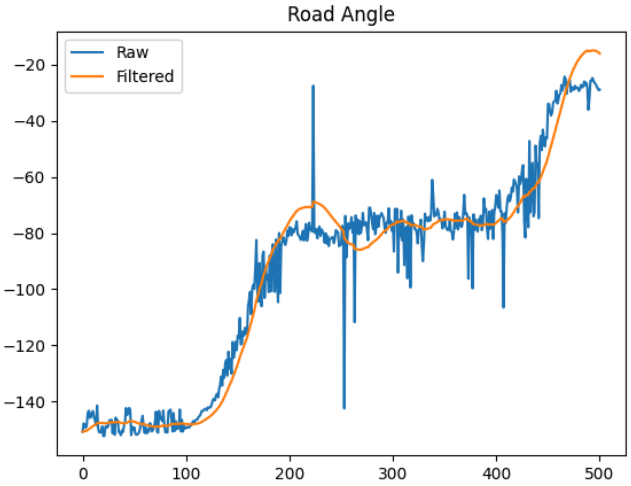
The complete code for our project can be found in our GitHub repository. Our algorithm uses the steps listed below. The methods that correspond to each step are mentioned in brackets.

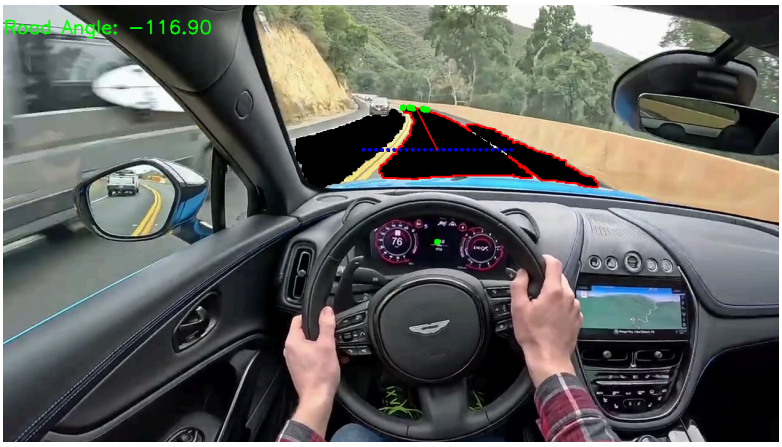
1. **Initializations**:
    a. **Feature Tracking Initialization [init_feature_tracking]:** Initializes the ORB feature detection later used for optical flow as well as OpenCV's brute force matcher to match those features across frames.
    b. **Template Loading [load_template_pyramid]:** Loads in a feature template and creates a template pyramid from the original template for better matching.
    c. **Video Capture Loading:** Loads in the training video, sets the FPS, and sets the frame number from which to train/evaluate the model.
2. **Find Speedometer [find_speedometer]:**
    a. **Template Matching [find_speedometer_template_matching]:** This method applies template matching to a template pyramid to find the best match. If the best match is below a certain threshold (meaning no good match was found), it returns None.
    b. **Template Matching Trust Evaluation:** To fight noise, the trust evaluation evaluates how reliable the results of the template match are by comparing them to previous matches. If the new match is part of a series of x successive matches and all of those matches are around the same area, the result of the template match is considered "trusted."
    c. **Optical Flow Vector Calculations:** This section begins by masking out the "outside world" through the car's windshield by using the last known location of the speedometer as a reference. This is because we don't want the movement happening outside the vehicle to impact our calculations. Next, it applies the ORB feature detector to match a set of "keypoints" in the image. These keypoints are then compared to those from the previous frame using a brute force matcher (to calculate which features from the previous frame map to which features in the current frame). Then, it calculates a difference vector to measure how each keypoint has moved since the last frame. Finally, after filtering outliers, the average of all these difference vectors is taken to yield the average optical flow of the image (the "mean_vector").
    d. **Optical Flow Prediction:** If the result of the template matching was considered "trusted," it discards the optical flow calculations (the reason we still did the previous steps is to account for the case where the next frame needs information from the current frame, though admittedly there is some room for minor optimizations here). However, if the template matching either fails or is not considered "trusted," it uses the average optical flow vector as calculated in the last step and offsets the previous prediction by the "mean_vector." This results in a new prediction for the speedometer's location based on the video's optical flow.
    e. **Alpha-Beta Filtering [AlphaBeta#update, AlphaBeta#predict]:** The speedometer prediction can still be a little noisy at this point so we apply Alpha-Beta filtering to the predicted output yielding the final result. If for some

reason, both the optical flow prediction and template matching fails to yield a result (which is extremely rare), we use the past history of predictions and the Alpha-Beta filter to predict an output.

3. **Get Road Angle [get_road_angle]:**
    a. **Offsetting & Flood Fill Seed Initializations:** Uses the location of the speedometer as a reference point to create seed points for a flood-fill algorithm above the speedometer. We assume the constant values of these offsets are given as hyperparameters per vehicle. Creating multiple seed points, as visualized below in blue, allows our road segmentation to account for *some* variation in colors on the road. The flood-fill result of each of these seed points is then combined in a single mask. Finally, this combined mask is dilated before we do contour matching to kill the noise in the flood-fill results. This yields a relatively robust segmentation result.
    b. **Contour Detection:** Taking the combined flood-fill mask as an input, we then do contour detection on the mask of the road to extract coordinates for how the road is segmented.
    c. **Angle Calculation:** This step uses the contours detected in the previous step to find the topmost pixel. Then, to make stable and reliable predictions, we find all pixels that are vertically within 5 pixels of the topmost pixels. Then, using these pixels, we find the average horizontal position of these topmost pixels. We now get an average of the x positions and take the y position of the topmost pixel. Finally, starting from our middle seed point, we draw a line to this "top position" as calculated (see the below image for a visualization). This line represents the "angle of the road" that we want to measure. Indeed, we can now finally solve for this angle using the following formula (derived from some basic trigonometry):
    $\theta = \tan^{-1}(\frac{y2-y1}{x2-x1})$

d. **Alpha-Beta Filtering [AlphaBeta#update, AlphaBeta#predict]:** The angle output from the previous step is still a bit noisy. Thus, we apply the Alpha-Beta filter to denoise our measurement. The results of this step are also shown below.



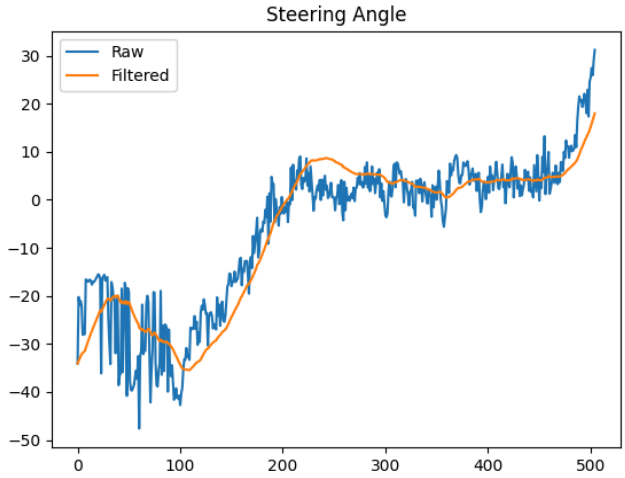


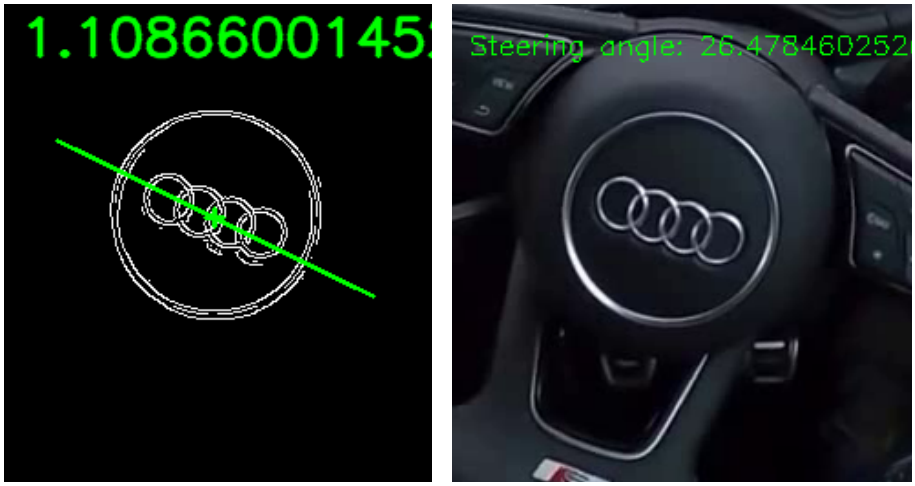
4. **Get Steering Angle [get_steering_angle]:**
    a. **Cropping to Get Region of Interest (ROI):** Given the detected speedometer area as input, we crop the frame to include that rectangle and everything below it.
    b. **Canny Edge Detection:** Next, we convert the extracted ROI to grayscale and apply Canny Edge detection to the grayscale image to detect edges.
    c. **Dilation:** We then apply dilation to the detected edges using a 5x5 kernel to enhance them.
    d. **Contour Detection & Extraction:** In this step, we find contours in the dilated edge-detected image. Then, we find the largest contour among the detected contours based on contour area. Next, we create a mask image for the largest contour and apply the mask to the edge-detected image to isolate the edges within

the largest contour. This results in an image consisting only of the car manufacturer's logo.

e. **Steering Angle Calculation:** First, we calculate the area of the logo and find the centroid coordinates. Next, we compute the moments of inertia components a, b and c. Based on these moments, we calculate the angle of least inertia θ using the formula: $\theta = \frac{1}{2}\tan^{-1}(\frac{b}{a-c})$. We then draw a line representing the axis of least inertia and a marker at the centroid on the image for visualization. Finally, we convert the angle of least inertia to the steering wheel angle using the following formula: $(90° - |\theta|)$ if θ is positive and $(|\theta| - 90°)$ if θ is negative. This is done because the Audi logo is horizontal, so the angle of least inertia is 90° when the steering is straight. By doing this conversion, the steering angle is returned as 0° when the steering is straight. If a manufacturer has a vertical logo, then we do not need to account for the logo orientation being perpendicular to the steering angle, and the calculation is simpler.

f. **Alpha-Beta Filtering [AlphaBeta#update, AlphaBeta#predict]:** The output from the previous step is noisy. Moreover, sometimes the contour extraction may fail. Thus to deal with this, we use an Alpha-Beta filter once again to denoise the output, and in the cases where we are unable to get a measurement, we predict angle using data from previous measurements. Result of this denoising step is visualized below.





5. **Support Vector Regression Modeling**:
    a. **Model Data & Feature Set:** Our model is based on the observation that the points in time at which the angle of the road influences the angle of the steering wheel are offset — i.e., there is a delay between measuring the angle of the road and having to turn the steering wheel to adjust for it. This is visible as a phase shift between the road angle curve and the steering angle curve in the graph above. However, this phase shift is not constant (we theorize it is proportional to vehicle speed), although it does average to around 40 frames. To account for this, for each sample, we take a snapshot of the past N=100 past road angles and feed that into our model as its feature set. This gives the model access to the entire time series of road angles, allowing more sophisticated modeling of how angle over time affects vehicle steering.
    b. **Model Design:** The model itself is a support vector machine-based regression model, which we built using the Python library scikit-learn. SVMs are a powerful tool that allows us to fit a curve to very irregular and noisy data, which is what we see here, which is why we thought it would be an appropriate model to use. Our feature set consisted of length 100 vectors consisting of a time series of the prior 100 road angle samples, while our target data was a scalar value representing the "correct" steering angle based on our segmentation results. While we did ostensibly say we were using classical methods, and SVMs are arguably a form of supervised machine learning, we do not feed the image directly into our model, so we have stuck to our design goals.

Experimentation and Results

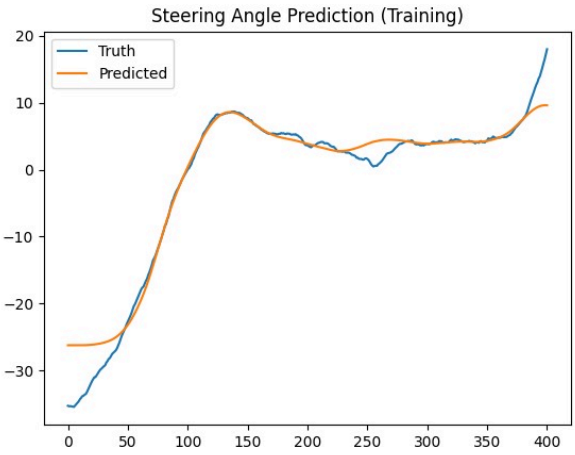
Evaluation Metrics

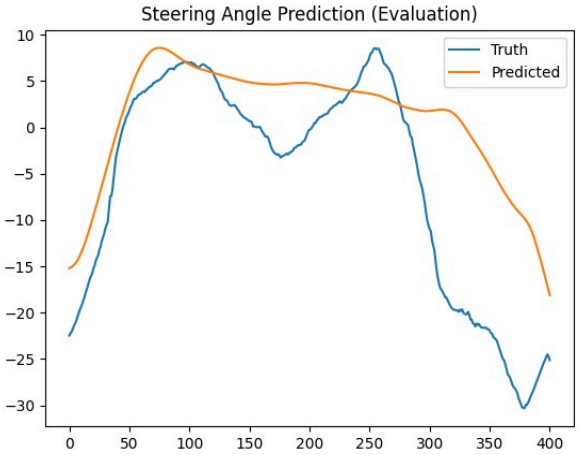
Our primary evaluation metric for our segmentation algorithm was visual, i.e., drawing in the various segmentation steps in OpenCV and playing the segmented video back to the user. Since the segmentation process is based entirely on classical vision and is therefore deterministic, visually inspecting the segmentation result was largely a good indicator of whether or not segmentation was successful. However, we were also mindful in evaluating the noise that various segmentation methods introduced. For instance, an early prototype of our steering wheel segmentation locked onto the hands of the driver rather than the wheel itself — it worked reasonably well but caused significant quantifiable noise as the driver moved their hands.

We evaluated our model using the root mean squared error (RMSE) between our predicted values and the "true" values based on steering wheel segmentation, which is an effective way of evaluating the effectiveness of a regression model.

Results

Our model, when run on its training data, had an RMSE of 2.37 degrees, but when run on a previously unseen evaluation dataset, the model achieved an RMSE of 9.88 degrees, which is not great. This indicates that our model is significantly overfitting, likely due to the limitations of using road angle alone and the relatively limited amount of training data. Note that the steering angle prediction that was not overfitted was able to capture the overall shape of the data but not its finer details, unlike the original model, which was overfitted but captured considerably more detail





However, our segmentation system works very well. It allows us to gather good data based on first-person video footage and is fairly tolerant to inconsistencies in roads, and noise in the video of the dashboard such as hand movement, windshield wipers, subtle lighting changes, etc.

Runtime Performance

Our code is intended to provide real-time predictions. As such, performance of our code is important. When not in "debug mode" we calculated our code took about 139 ms to compute on average per frame on our training dataset. This yields an average frame rate of 7.2, which while not impressive is still feasible for real-time use.

Analysis of our runtime shows that by far the most expensive part of our method is the template-matching across pyramids. These results show that there is certainly room for improvement here especially by instead trying to use localized template matching through a more clever template matching scheme. However, our preliminary attempts to achieve this also negatively affected our model results.

Name	Call Count	Time (ms) ▼		Own Time (ms)	
main.py	1	148733	100.0%	1	0.0%
run	2	146331	98.4%	2475	1.7%
find_speedometer	1006	113954	76.6%	2579	1.7%
find_speedometer_template_matching	1006	97231	65.4%	385	0.3%
<matchTemplate>	4024	95319	64.1%	95319	64.1%
get_road_angle	1006	23150	15.6%	6341	4.3%
<floodFill>	25050	11255	7.6%	11255	7.6%
<method 'detectAndCompute' of 'cv2.Feature2D' objects>	1006	10995	7.4%	10995	7.4%
<bitwise_or>	25050	2918	2.0%	2918	2.0%
get_steering_angle	1006	2513	1.7%	36	0.0%
<method 'copy' of 'numpy.ndarray' objects>	6002	2405	1.6%	2405	1.6%
<method 'match' of 'cv2.DescriptorMatcher' objects>	1004	2188	1.5%	2188	1.5%
<method 'read' of 'cv2.VideoCapture' objects>	1006	2156	1.4%	2156	1.4%
_find_and_load_unlocked	1291	1949	1.3%	5	0.0%
_find_and_load	1315	1949	1.3%	9	0.0%
_load_unlocked	1245	1941	1.3%	5	0.0%
exec_module	1040	1941	1.3%	3	0.0%
_call_with_frames_removed	1735	1936	1.3%	1	0.0%
<built-in method builtins.__import__>	349	1626	1.1%	1	0.0%
find_angle_of_least_inertia	1006	1582	1.1%	666	0.4%

Discussion

Strengths

Our road segmentation system is able to robustly segment the road using the flood fill algorithm based on the location of the dashboard in the camera view. This allows it to function under various road conditions, and is even somewhat tolerant to changes in road conditions, although we still assume approximately constant lighting in our test data.

Our dashboard recognition system and measurement of the steering wheel is also very effective, and is able to get a good measurement of the steering wheel angle irrespective of the driver's hands or current road conditions, as long as the steering wheel is in frame.

Limitations

Our model suffers considerably from overfitting, with a ~7 degree difference between the RMSEs of the training and evaluation sets. This is primarily due to the limited availability of suitable data and to having only one real feature to input to the model, road angle. While our model consists of 100 "features," each representing a point in a time series, these are only derived from a single measurement, limiting the amount of information our model can access. If we were able to successfully measure car speed based on the OCR of the speedometer, we would have more data to work with and perhaps be able to alleviate this issue since the phase shift between the road angle and steering angle varies considerably based on speed. However, road conditions also affect the phase shift, so this would not be a complete solution.

We also have a recurring problem with noise in our data. While the Alpha-Beta filter helped considerably to reduce the issue, it still persists on some level and could be further mitigated with more sophisticated filtering algorithms.

Areas of Future Work

It would be very useful to be able to measure vehicle speed for inclusion in our model, as mentioned above. This could be done by OCR on the car dashboard speedometer, as most modern cars have digital speed readings. Measuring the angle of an analog speedometer could also be useful, albeit considerably more challenging. In fact, we did attempt to incorporate OCR into our model. However, off-the-shelf methods are too slow for a real-time software like ours. More research would be required to implement a fast yet robust digit-OCR system.

We would also like to continue working on our current model by including more diverse training data and similarly diverse evaluation data. The entire model was trained and evaluated based on short drives on an empty road.

Conclusion

All in all, we have made a fairly decent "scale model" of a self-driving car. Our model leaves much to be desired, primarily due to a lack of features, but we are very happy with our segmentation results, especially considering the results achieved without the use of machine learning. This is especially the case because the segmentation algorithms we developed could be used to develop larger self-driving car datasets based solely on footage, using incredibly

accessible tools to collect numerical data on how a vehicle is driven: just a camera and a motorcar. Moreover, with no machine learning, our algorithm can also be run on most consumer hardware, which is a significant plus of what we achieved.

Credits and Bibliography

“2020 Audi Q2 30 (6MT) - POV Scotland Drive to Skyfall (Binaural Audio).” YouTube, YouTube, 24 Dec. 2020, www.youtube.com/watch?v=GHLK0gXEKxU.

“2024 Aston Martin DBX707 - Pov Canyon Blast (Binaural Audio).” YouTube, YouTube, 28 Mar. 2024, www.youtube.com/watch?v=gdG35Ame-uY.

“Contours: Getting Started” OpenCV, https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html

E. Rublee, V. Rabaud, K. Konolige and G. Bradski, "ORB: An efficient alternative to SIFT or SURF," 2011 International Conference on Computer Vision, Barcelona, Spain, 2011, pp. 2564-2571, doi: 10.1109/ICCV.2011.6126544.

Nayar, Shree, lecturer. Geometric Properties | Binary Images. First Principles of Computer Vision, Youtube, 1 Mar. 2021, https://www.youtube.com/watch?v=ZPQiKXqHYrM.

“Optical Flow.” OpenCV, docs.opencv.org/4.x/d4/dee/tutorial_optical_flow.html. Accessed 25 Apr. 2024.

“OpenCV Features2D Draw.Cpp.” GitHub, github.com/opencv/opencv_attic/blob/master/opencv/modules/features2d/src/draw.cpp. Accessed 25 Apr. 2024.

“Template Matching.” OpenCV, docs.opencv.org/4.x/d4/dc6/tutorial_py_template_matching.html. Accessed 25 Apr. 2024.