

TRITON EYE



Project Lead - Sameer Kausar

Backend Engineer - Yeseong Kim

Frontend Engineer - Pranay Pant

CSE 145/237D Professor Ryan Kastner

Spring 2016

Final Report

Repository: <https://github.com/yeseongk/tritoneye>

Abstract

Even though a number of infrastructures are available to count the number of vehicles, they require a high amount of costs to deploy. One promising way to reduce the deployment cost is to leverage a computer vision-based vehicle detection on emerging small computing devices such as Raspberry Pi. In this project, our system exploits a network of Raspberry Pi's with cameras equipped on each floor and counts the number of cars based on OpenCV library. The output is sent to a server over TCP/IP protocols and users can access the information through a website. In our evaluation conducted on a field testing, we show that our systems can report the available parking spots to users on a website by efficient video processing on Raspberry Pi.

1. Introduction

We are creating system that provides live parking availability. This will allow for individuals looking for parking to spend less time searching for spots. This is especially relevant today since most commuter students, faculty and UCSD employees do not have the time to circle around campus hunting for parking. In addition, the solution should display the parking count in a user-friendly way, such as live displaying on a website.

In this project, we developed Triton Eye, which utilizes multiple camera-equipped Raspberry Pi's and a web server that assimilate the information received from the Raspberry Pi's to server the information to users via their web browsers. The system uses a network of cameras (one per floor) to count the number of cars arriving and leaving. There are many detailed strategies that have been published but we will design an algorithm that is optimized for UCSD and Raspberry Pi to guarantee reasonable local processing time. The cameras would be run on a *Raspberry Pi* and send only the processed data. In other words, the camera will capture the video and the *Raspberry Pi* will process data locally and send an output (ie, car came or left). We would be using *OpenCV* and other open source libraries to create an algorithm to count cars and determine what types of spots are open (A, B, S, etc). With this output sent to a server (using a TCP/IP protocol with some sort of encryption), a website will display the live count of cars per parking lot/structure. The server first performs a parking lot counting algorithm which computes the available number of lots of each floor by using the car counts and the directions received from the Rapsberry Pis. The information is stored into a Sqlite database, thus we can retrieve the real-time parking spot information and show it to the web user interface running on Ruby on Rails.

This project is split into 3 main technical parts. The backend car counting algorithm run on the Raspberry Pi, the frontend and network communication between the Pis and the website and finally the open spot algorithm that takes the output of the car counting algorithm and converts it into something the user wants. Yeseong, the backend engineer, focused on creating a lightweight algorithm to run locally on the Pi that accurately counted the cars coming to and from using OpenCV libraries. Pranay, the frontend engineer, focused on the communication between the server and the Pis and created the website that displays the information. Sameer, the team lead, focused on the layout of how the Raspberry Pis would be placed and from their placement, developed an algorithm to determine the number of spots open. He also made sure the team was on track with their goals and ensure the quality of each part.

We implemented the Triton Eye on Rapsberry Pi 2+ for the backend and a laptop for the frontend. For the Raspberry Pi side, our program implemented in Python is running with OpenCV 3.0.1, and the webserver that executes the parking spot counting algorithm and serves the web interface is running on Ruby on Rails. We also evaluated our solution in a in-lab testing that uses custom videos with moving toys and stocked videos found in Internet. Then, we conducted two field testings in Pangea and Hopkins structure. In the first testing, we found the pracissues such as a headlight reflection, and in the second testing, we verified that our improved algorithm could address the practical issues in most cases. Our future plan is to improve our algorithm to minimize the false positive countings and to eventually create an iOS/Android app or integrate it with the current UCSD app.

2. Technical Material

2.1 Overview of Triton Eye

In this section, we describe our parking spots counting solution for UCSD parking structures, named Triton Eye. There are two key goals in designing our approach. First, the actual deployment cost of systems should be minimal compared to the existing sensor-based solution which requires lots of sensors for every parking spot. Second, the number of available parking spots should be easily accessible by users, and the information needs to be updated on time. In order to achieve the first goal, we designed our systems running on a Raspberry Pi with a camera which is required to be deployed on each floor. Thus, the technical challenge for this is how to process the real-time video stream in a performance effective way. For the second goal, we have developed a web server-based system, which provides user interfaces in a webpage to show the aggregated information for Raspberry Pi's in a parking structure. The webpage has been designed so that it can retrieve the information from the video processing results of Raspberry Pi's over the network communication.

Figure 1 shows an architectural overview of the developed Triton Eye and Table 1 also shows the development environment used in our implementation. Triton Eye systems consist of two parts: i) multiple backends running on Raspberry Pi to process real-time video and count the number of cars arriving and leaving; ii) a frontend running on a web server running using Rails which aggregates the information of car counts for a parking structure, performs a parking spot counting algorithm, and eventually shows the information to the user via a web page. In the following subsection, we describe the technical details of Triton Eye, the video processing methodology of the backend side in Section 2.2, the parking spot counting algorithm in Section 2.3, and the web server development in Section 2.4.

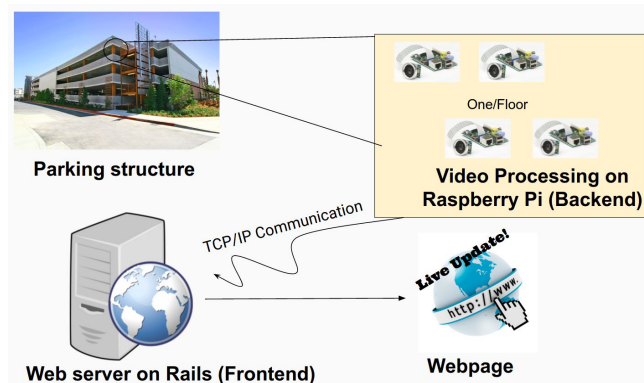


Figure 1. An architectural overview of Triton Eye

Table 1. Development environment and requirement

	Hardware	Software	Note
Backend	Raspberry Pi 2+ Camera	OpenCV 3.0.1 Implemented on Python	- TCP/IP communication to the frontend - Developed traffic log reproducer to simulate a backend
Frontend	Any web server. Implemented on a laptop.	Ruby on Rails Sqlite for database	- Include an admin page to configure the floor

2.2 Vehicle Counting on Raspberry Pi

The backend program running on Raspberry Pi counts the number of cars with the direction of each car so that it can report the related information for each floor to the web server in a realtime manner. The vehicle tracking procedure of the video processing algorithm performs two steps, an object identification and an object tracking. We have used OpenCV 3.0.1 library to process the real time video, and all the source code are developed in Python 2.7. On the top of that, since we only have a single Raspberry Pi, in order to simulate multiple Raspberry Pis which communicate to the frontend web server at the same time, we develop a backend traffic log reproducer which generates network packets based on recorded car counting logs in the same way that the original Raspberry Pi does.

2.2.1 Object Identification

The first part of video processing algorithm is to identify moving objects in a video. Unlike other high-performance computing devices, the Raspberry has limited capability of the processing power. Even though there are several algorithms developed to track the number of moving objects by capturing their distinguished features such as Haar Cascade algorithm [1], in our testing, we found that the algorithms can not process the real time video due to the limited performance of the Raspberry Pi. Thus, we developed our own procedures to find moving objects in a video.

The object identification step first applies a background subtraction algorithm. As the OpenCV library provided, there are three popular algorithms, called MOG [2], MOG2[3], and GMG [4], where all identify changed pixels by considering consecutive frames to subtract the background in a video stream. We found that the three algorithms provide sufficient performance to process the video. According to each algorithm, we can process more than 10 frames for a real time video taken by camera, which is enough to identify moving objects in a frame. In addition, an advantage of the MOG and MOG2 algorithm is that they can distinguish the shadow from the actual objects. We developed our procedure so that the selection of each algorithm is configurable. However, since we also empirically found that, in general, the MOG2 provides the best accuracy in finding the moving objects including vehicles, we use the MOG2 algorithm in our field testing.

After the background subtraction procedure, the processed frame contains white pixels which represents the moving pixels, while others shows black pixels which are identifies as the background. However, the frame processed by the background subtraction algorithm is often too sensitive, so containing lots of noisy pixels scattered in a small arare. Thus, to remove the pixel noise from the frame, we further applied two filters sequentially: opening and median filter. The first opening filter dilates every pixel so removing small pixels from the frame. Then, the median filter is also used to merge disconnected parts, which are close enough, since they are likely to be included in a single object.

The last procedure of the object identification is to find contours of objects in a frame. Since a frame may include multiple moving objects, it distinguishes different objects in a frame, where an object is assumed to have one contour. Then, for each identified contour, the algorithm processes two post-processing work to execute the algorithm more robustly. The first one is to eliminate either too small contours or too large contours which can not be a vehicle, and the thresholds are also configurable depending on the camera position. The second step is called as sudden frame change identification which checks how many pixels are changed in a entire frame. Once it occurs, we ignore the frame and reset the background subtraction algorithm since the camera sometimes produces such images when either intensive light changes or physical camera movements happen.

Figure 2 shows the result of the object identification procedure for a stocked video found in Internet. The first frame shows the original image from the video where the green color lines show contours of each identified vehicle and the red color box shows its minimum area. The second and third frames present its internal procedure to recognize the moving objects. As shown in the second frame, the background subtraction can find the moving pixels with small noise pixels, and it is successfully filtered out in the third frame.

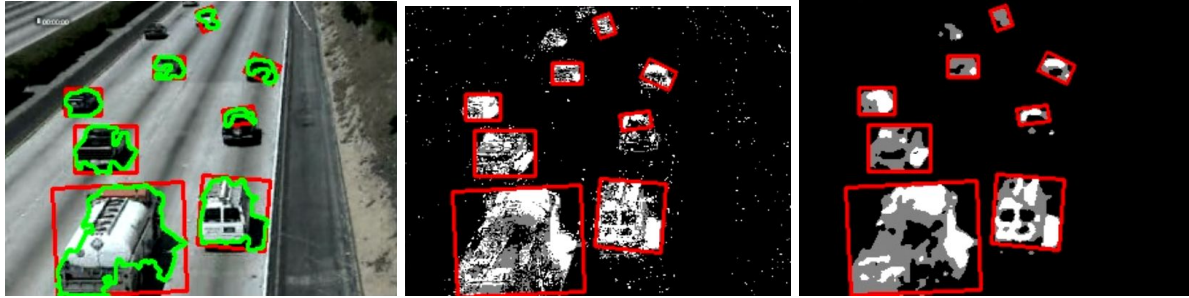


Figure 2. Object identification procedure (original, background subtraction, and filtering)

2.2.2 Object Tracking algorithm

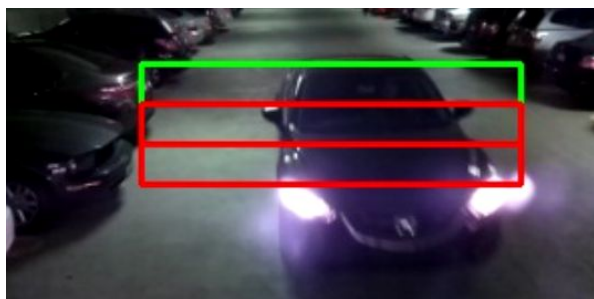
To compute the number of cars with their directions, we developed two algorithms that exploit different criteria. The main reason why we need different methodologies is that we need to handle videos taken on different camera angles and different subsequent environments. For example, if the camera is installed in the top-bottom view, the camera takes videos that contain the headlights which unlikely happens in a horizontal camera view. Thus, we used a line-based tracking algorithm for the horizontal view, while an area-based tracking algorithm is used for the top-bottom view.

The line-based tracking algorithm utilizes a given reference line to count the number of cars which cross the line. The line is configured by clicking the video screen in the Raspberry Pi program. Figure 3(a) shows an example how we count the number of cars with their directions in the line. For each identified contour for two consecutive frames, we first compute which one is the same object over the frames. To this end, we compute the center of mass of each object and the distance between two pairs of objects which are separately in the two frames. Then, the closest object pair, which are both recognized in two frames, creates a single car that have the same ID (denoted as blue color text.) Then, we can compute their paths of each car as denoted by red color lines. The path of each car is compared against the given reference line, the yellow color line to check if the center of mass of each object crosses the line. In actual computations, the line splits the 2D space of the frame into two parts, and we check where the center of mass belongs, so estimating the correct direction of each car.

The area-based tracking algorithm utilizes an area of interest which is also configurable in the same way of the reference line case, as shown in Figure 3(b). The area of interest is divided by three subarea, and we compute whether each subarea is occupied by the recognized objects. Then, by considering the occupancy of the three area over subsequent frames, we can recognize the direction of cars as well as the number of cars. Since we estimate the direction of the car without knowing actual paths of the moving objects, we could handle the car headlights and their reflection on ground, which is hard to distinguish in the line-based tracking algorithm.



(a) Line-based tracking



(b) Area-based tracking

Figure 3. Examples of two vehicle tracking algorithms

2.2.3 Network Communication and Traffic Log Simulator

The computed information for the number of cars and their direction at a moment is sent to the web server over TCP/IP communication. The transmitted packets are also logged, i.e., while the actual backend program processes either a recorded video or a live camera stream, it can also save the network packets, which will be transferred over TCP/IP communication, into a text file with each of timestamps. It is because we need to evaluate our entire system in the situation when more than one Raspberry Pi are connected. In order to reproduce the network packets, we also developed a traffic log simulator which generates packets at the same time actually processed in the real Raspberry Pi. The simulator was also useful to test our systems since we can speed up the simulation time.

2.3 Parking Spot Counting Algorithm

The algorithm used to count the number of parking spots utilizes the output of the Raspberry Pi and exploits the location at which the Pis are set up. Algorithm 1 shows a pseudocode of the algorithm. Each Pi is located on the hills between the floors and one on the entrance, which means it tracks two subsequent floors. This ensures that we know exactly what floor each car is moving to and from. The floor also includes a virtual entrance floor, which means outside to correctly the count number of cars coming into the first floor. The Pi sends a 1 or 0 dictating come or go along with its tag, as denoted CAR_ENTER and CAR_LEAVE in Algorithm 1. For example, if the Pi on the hill between the first and second floor, facing downhill towards the first floor sends a 1 it means that a car came towards the Pi aka going from the first floor to the second.

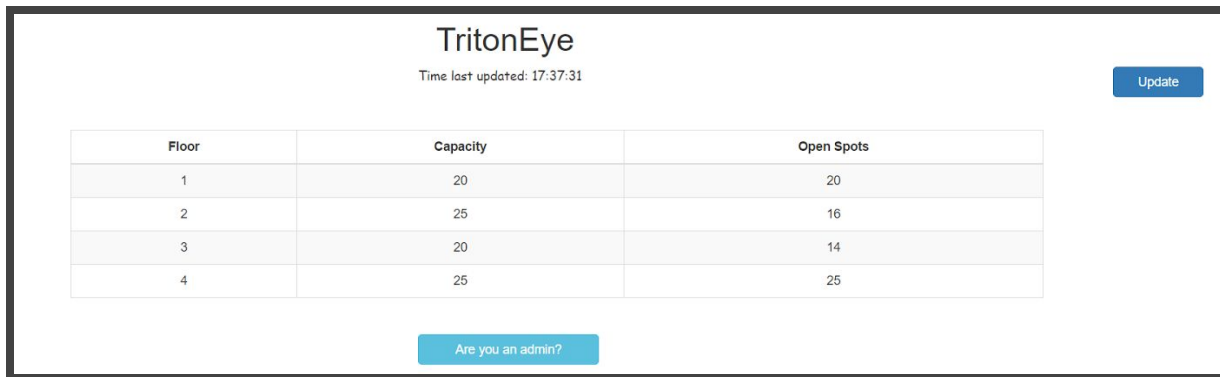
We use an array stored in a database that keeps the number of cars per floor and one to store the total number of spots available. Index 0 corresponds to the total number of cars in the parking lot and each index holds the number of cars on that floor. The admin defines the spots available array when inputting their data into the form we provide on the website. After getting the signal from the Pi, the algorithm maintained two variables for each floor to track the number of the car, *cars* variable, and the total open spots, *open_spots* variable. We update the total number of cars in the parking lot array, *find_spots()* in Algorithm 1, and then take the difference from the capacity and 0 to properly predict how many spots are open while it doesn't give a negative number or a wrong number greater than the total capacity, *store_adjusted_openspots()* in Algorithm 1. That is, since the number of cars of a floor can exceed the number of open spots when a car is entering when the plot is full, we consider this case to show the correct number in the website. We also utilize the total car count to do sanity checks and to account for driver lag. For example, if there is only one spot left on the third floor and a car enters the structure, it takes time for it to reach the third floor and count the car as being there. If another car come in after, the third floor is now full but since the car has not hit the sensor yet, the algorithm would not count it. To account for this, we look at the total number of cars in the structure, which is being stored at index 0 and is updated everytime a car enter/leaving the structure. From there, we look at the open spots on each floor and make a guess about where the will park to account for the lag time it takes to drive up the structure.

Algorithm 1. A pseudo code of parking spot counting algorithm

<pre>function find_spots (f1, f2, direction) if (direction == CAR_ENTER) f1.cars = f1.cars - 1 f2.cars = f2.cars + 1 end if (direction == CAR_LEAVE) f1.cars = f1.cars + 1 f2.cars = f2.cars - 1 end store_adjusted_openspots(f1) store_adjusted_openspots(f2)</pre>	<pre>function store_adjusted_openspots (floor f) cur_spots = f.capacity - f.cars if (cur_spots > f.capacity) cur_spots = f.capacity end if (cur_spots < 0) cur_spots = 0 end f.open_spots = cur_spots save_to_db(f)</pre>
--	---

2.4 Dynamic Web-based User Interface

The web server program has two main parts: the algorithm that takes data from the Pi and calculates the number of spots available, described in Section 2.3, and the actual website that displays the information to the user. The website itself is built using Ruby on Rails. The website has two components - the Rails app framework, which create the website, and an server file written in Ruby, run within Rails environment but separately from the app. The server code is what allows the data to be collected from the Pi. Since the server code is run from within Rails, it has access to the same libraries that website code does. Thus, it can call the parking spot algorithm, passing in the parameters necessary. This is important, since the only place we have the data from the Pi is within the server code. This makes it easy to use data from the Pi, and stores/transfers the retrieved parking spot information in the same database deployed in Sqlite.

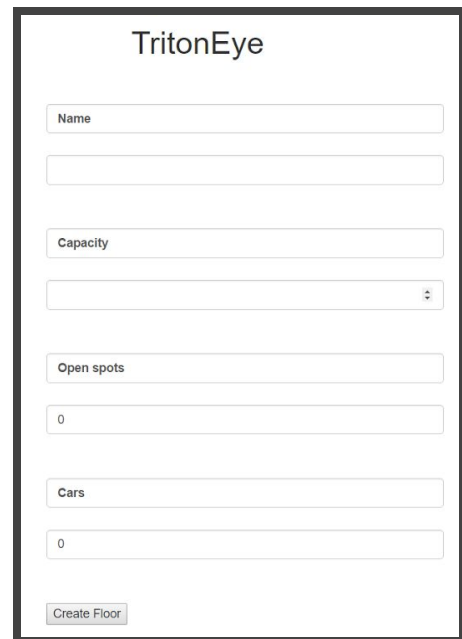


The screenshot shows the TritonEye main web page. At the top, it says 'TritonEye' and 'Time last updated: 17:37:31'. There is an 'Update' button in the top right corner. Below this is a table with three columns: 'Floor', 'Capacity', and 'Open Spots'. The table contains four rows of data. At the bottom of the page, there is a button that says 'Are you an admin?'.

Floor	Capacity	Open Spots
1	20	20
2	25	16
3	20	14
4	25	25

Figure 4. User interface in the main web page

Once we have collected data from Pi and passed it to the algorithm, all that is left is displaying it. This is done utilizing Rails's MVC architecture - the root or home page is the index page of the parking lot, where all the cars for each floor are shown. The actual user interface of the main page is shown in Figure 5. There is an option to navigate to a special page, shown in Figure 6, if you are an admin and input data for a new parking lot, then redirect back to this home page. Lastly, all of this is styled using twitter-bootstrap for user appeal. In this way, we see how the data collected from the Raspberry Pi about cars entering/exiting can be used to calculate how many available spots a parking lot has and show that info to the user in a clean manner. It also provides the time information when the page is updated, so that the user can update using the button on the top of the page.



The screenshot shows the TritonEye admin page. It has a form with several input fields: 'Name', 'Capacity', 'Open spots', and 'Cars'. There is also a 'Create Floor' button at the bottom. The 'Open spots' and 'Cars' fields have the value '0' displayed.

Figure 5. User interface for the admin page

3. In-lab and Field Testing and Deployment

3.1 Testing Methodology and Results

We tested our methodology in two ways. Before going to the actual field test, we evaluated and optimized our program using videos of balls and toys as well as stocked videos to make robust algorithms. Figure 6(a) presents the result of this in-lab testing. In the in-lab testing, we found that the auto adjustment feature of the camera, e.g., exposure, makes instable video in that a large amount of pixels keep changes for some cases, e.g., according to the time of day changes. Thus, we added an additional feature that fixes the adjusted features on the our backend program.

Figure 6(b) presents a picture of the field testing. For the field testing, we prepared a Raspberry Pi which communicates over tethering via a smartphone and is powered by a laptop USB. To control the Raspberry Pi without actual monitor, the Raspberry Pi is also connected in the smartphone using the VNC protocol. In the first field testing conducted in Pangea parking structure, we found that our algorithm could successfully recognize and track the moving vehicles in most cases. However, the most challenging part which we didn't think is that the car has headlights and it can be reflected, as discussed in Section 2.2.2. In order to handle this issue, we recorded the field videos that makes the program, and implemented the area-based counting algorithm. Then, we conducted the second field testing in Hopkins parking structure, and verified that the improved algorithm can address many practical issues. The evaluated results of the backend program can be found in the youtube links below:

- Line-based tracking
<https://www.youtube.com/watch?v=LINH5eP4T3M>
- Area-based tracking
<https://www.youtube.com/watch?v=K-KsFMKGvdQ>

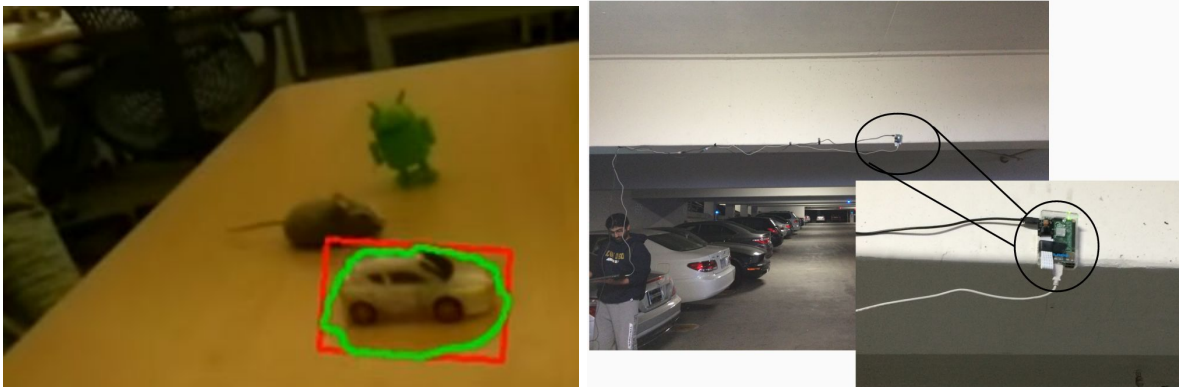


Figure 6. Left: in-lab testing (a), Right: field testing (b)

3.2 Triton Eye Deployment

All the developed source code are available in the github with a description, and it includes following implementation:

Table 2. Triton Eye repository, description, and usage

https://github.com/yeseongk/tritoneye		
Component	Directory	Description
Frontend	/	- Requirement: Ruby on Rails - Usage: (server) \$ rails runner server.rb (web app server) \$ rails s -b [IP_ADDRESS or 0.0.0.0 for generic] -p [PORT NUMBER]
Backend (Raspberry Pi 2+)	/src	- Requirement: OpenCV 3.0.1 - Usage: (Actual) \$ python triton_eye.py [-v VIDEO_FILENAME] (Simulation) \$ python simulation_triton_eye.py -sl LOG_FILENAME -ss SIMULATION_SPEED
Test code for OpenCV	/test_src	- Usable to check the valid setup of Raspberry Pi
Sample video	/sample_video	- Include stocked videos and field testing video recorded in UCSD
Sample log	/sample_log	- Input files for the traffic simulator - Recorded from the field testing sample video

In order to change the configuration in the deployment, the backend provides multiple options in the “src/recognition/recognition_conf.py”:

Table 3. Triton Eye backend program configurations

Processing procedure	Option name	Description
Object identification	BG_SUBTRACTOR	Background subtraction algorithm (MOG, MOG2, GMG)
	FILTER_NOISE_SIZE	Noise pixel sizes to ignore
	MIN_OBJECT_AREA MAX_OBJECT_AREA	Size of area to be determined as a single object
Object tracking	DETECTION_METHOD	Selection between line-based and area-based algorithm
	MIN_DISTANCE_ON_LINE	Pixels that determines if an object is on the reference line (for line-based algorithm)
	OCCUPIED_RATIO	Ratio that determines if an area is occupied (for area-based algorithm)

4. Milestones

Milestones	DONE? (Explained section)	Description
Milestone 1: Project setup and initial in-lab development (04/15~04/28)		
Object tracking algorithm	DONE (2.2.1)	Implemented the object identification algorithm
Application counting moving toys	DONE (3.1)	Tested with toys
Initial web interface with runtime update	DONE (2.4)	Created Interface was
In-lab testing with sample videos	DONE (3.1)	Tested with stocked video
Milestone 2: Field development (04/29~05/13)		
Field data recording	DONE (3.1)	Found the headlight reflection issue
Object tracking library improvement to handle overlapped objects	DONE (2.2.2)	Solved by line-based counting
Backend communication application	DONE (2.2.3)	Developed the traffic simulator as well
Web interface for single device	DONE (2.4)	Checked with communication methodology
Integration and testing using field video	DONE (3.1)	Found the headlight reflection issue
Parking spot identification algorithm (Optional)	NOT DONE	Supposed to be optional, and time didn't permit
Milestone 3: Tuning and Application integration (05/16~05/27)		
Image processing library tuning	DONE (2.2.2)	Implemented the area-based algorithm
Parking spot counting algorithm	DONE (2.3)	Implemented and ported into the web server
Field data recording 2	DONE (3.1)	Did the field testing twice to handle light issue
Web interface for multiple devices	DONE (2.4)	Add admin pages as well
Encryption for interface	NOT DONE	Time didn't permit
Parking spot algorithm integration to backend application (Optional)	NOT DONE	Supposed to be optional, and time didn't permit

We achieved most of milestones as we planned, but there are several things, either what we couldn't cover or what needs to be improved. Initially, we planned to implement an algorithm to identify parking spot types such as A, B and V in the UCSD parking structure case if time permits. However, since we required to handle the problems found in the first field testing as a higher priority, we implemented area-based algorithm instead. In the frontend side, we didn't cover encryption and admin authentication development due to the limited time, but they are not the core

implementation and would be easily added in the further implementation. Even though we showed the prototype of Triton Eye and working demo, there are some feasibility to improve the systems. For example, some false counting issue are still remained in tracking vehicles, and it may need to provide some solutions to solve practical issues such as battery.

5. Conclusion

In this project, we developed a way, called Triton Eye, to save people time and stress when it came to parking. This solution will help everyone on campus more efficiently park which in turn will increase the productivity in things that actually matter. Our goal was to display a live count of open spots for different parking structures on campus. We have designed a working proof of concept and achieved the goal. We successfully developed algorithms to count the cars, determine the available spots and display them to our user. Future work on this project would be to improve the quality of the car counting algorithm to better deal with some corner cases, add more intelligence to the open spot algorithm to account for false positive/negatives and finally to make the website more user friendly and get it onto the UCSD app.

References

- [1] Haselhoff, A., & Kummert, A. (2009, June). A vehicle detection system based on haar and triangle features. In Intelligent Vehicles Symposium, 2009 IEEE (pp. 261-266). IEEE.
- [2] KaewTraKulPong, P., & Bowden, R. (2002). An improved adaptive background mixture model for real-time tracking with shadow detection. In Video-based surveillance systems (pp. 135-144). Springer US.
- [3] Zivkovic, Z., & van der Heijden, F. (2006). Efficient adaptive density estimation per image pixel for the task of background subtraction. *Pattern recognition letters*, 27(7), 773-780.
- [4] Godbehere, A. B., Matsukawa, A., & Goldberg, K. (2012, June). Visual tracking of human visitors under variable-lighting conditions for a responsive audio art installation. In American Control Conference (ACC), 2012 (pp. 4305-4312). IEEE.
- [5] Opening for noise handling:
http://docs.opencv.org/2.4/doc/tutorials/imgproc/opening_closing_hats/opening_closing_hats.html