Appendix H: Data Analytic Application – Parking place occupancy and vehicle/pedestrian counting

Contents

Abstract

Modern cities greatly benefit from monitoring of parking spot occupancy, as well as vehicle and pedestrian traffic, as these statistics allow the city administration to allocate taxpayer resources efficiently to address parking and traffic problems in the busiest areas. Parking and traffic monitoring currently requires limited and expensive human resources, and this project explores the possibility of performing this task using computer vision algorithms. The solution should provide accurate parking status and traffic counts (of various vehicle types and pedestrians), even in non-ideal conditions, while requiring as little human intervention as possible. Two programs have been developed based on well-established algorithms using open-source library as a reference to illustrate the feasibility of integrating analytic applications over the deployed cameras: a parking detection program and a traffic count program. The parking detection program detects parking spot occupancy based on the number of edges and luminance level of each spot. As for the traffic count software, it uses motion detection to identify objects of interest (pedestrians or vehicles), tracks them across the frame, and increments a count when they enter or exit the frame. For the parking detection software, the objectives have mainly been achieved: it operates very accurately (typical accuracy of 98.2%, worst-case of 90.9%), except in some very difficult conditions, and requires no human intervention after an initial set up period. The traffic count software has partially achieved its objectives: it requires no intervention after an initial set up, and its consistency and accuracy are very high (almost all results except one are above 90%), but it cannot yet identify all classes of vehicles, and it functions worse in the presence of shadows. These findings demonstrate that it computer vision can provide accurate parking status, and has the potential to provide accurate traffic counts with some improvements to the software.

1. Introduction

1.1 Motivation

In recent years, technological advances in the field of the Internet of things have permitted the automated collection of parking availability and traffic count statistics. However, these systems are often developed for specific environments with application-specific hardware. Academic research on parking detection focuses on applications in parking lots [1] and often uses sensors (magnetometers and ultrasonic sensors) [2]. Similarly, research on traffic count focuses on motorways [3] and often features specific hardware, such as physical devices present on the road, various types of detectors (infra-red, magnetic, radar, ultrasonic) [4] and infrared cameras [5]. In the cases where video cameras are used, ideal camera setup conditions are present: a top-down perspective with no obstructions, relatively high-resolution footage, and great lighting conditions, as shown in the demonstration video clips provided with the reference parking detection [6] and motion detection [7] programs.

It would be very interesting to expand parking monitoring and traffic counting to city intersections, especially in downtown areas. This information can be used to provide parking availability and traffic congestion to motorists, which improves their quality of life significantly: for example, due to the large number of cars, it becomes increasingly difficult to find available parking spots, so many individuals who cannot find parking stop in illegal locations and hinder traffic. Furthermore, it is very helpful for the municipality to have statistics on the amount of traffic in key zones, for both pedestrians and vehicles. Whether the objective is to determine which areas are the most at risk for congestion, which ones require improved infrastructure, or to identify which parts of the city attract the most visitors, having accurate traffic statistics allows the municipality to allocate revenue from public investment most efficiently. Unfortunately, limited resources often preclude the use of application-specific hardware: only ordinary IP cameras are available for this project because they will be used for multiple different applications. Furthermore, a top-down perspective cannot be achieved due to restricted mounting locations (causing issues with objects obstructing others and perspective), lighting conditions will differ significantly with time of day, and video quality will potentially be lower if the system is implemented en masse.

The goal of this project is therefore to verify the possibility of developing high reliability video analytics using multipurpose cameras in non-ideal camera locations. More specifically, it is to investigate whether computer vision can be used to achieve accurate parking status detection and gather an accurate count of vehicles and pedestrians in urban intersections, which are characterized by suboptimal camera placement and footage quality. To accomplish this, the objective is to develop software that can analyze a video stream of an intersection in an area of downtown Montreal and provide statistics on parking and traffic flow. For parking, it should detect which parking spots and illegal parking zones are occupied or free. For traffic detection, it should detect and distinguish pedestrians and vehicles of various classes, track them as they pass through the area, and count the number of each of them passing in each direction. Reliability should be higher than existing programs [6] [7], the software should function autonomously after an initial setup process to minimize the use of human resources, and provide reliable detection even in non-ideal weather conditions.

1.2 Background, related works

The platform chosen for this project is the OpenCV, an open-source computer vision library for the C++ programming language. It has the advantages of being free of charge, minimizing the overall development cost, and being feature-rich, implementing many state-of-the-art image processing algorithms. The parking detection software is based on the “Automatic Parking Detection” software developed by Github user eladj and modified by Mr. Nhat-Quang Dao [6]. This software’s intended function monitors a video stream of a few parking spot near a city intersection and detect if a given parking spot is free or currently occupied by a vehicle based on the level of detail in a given area (since asphalt is more uniform and appears smoother than a vehicle). The aforementioned software does not contain any function for traffic detection, so

that portion of the software was developed from scratch, with inspiration from the following open-source projects: "Motion Detection and Speed Estimation using OpenCV" by Hardik Madhu [8], "Basic motion detection and tracking with Python and OpenCV" by Adrian Rosebrock [9], and "Motion detection using a webcam, Python, OpenCV and Differential Images" by Matthias Stein [10]. The common aspect between these programs is that they perform object detection using motion detection, so the traffic detection software shall henceforth be referred to as "motion detection". They isolate moving objects using a difference frame, then (optionally) apply thresholding to improve definition of moving objects, and find contours in the resulting frame. Another program, "Vehicle Detection, Tracking and Counting" by Github user andrewssobal [7] also applies these techniques for traffic counting, so it will be the reference software to which this project's motion detection software will be compared.

1.3 Contributions

For the parking detection software, the following image processing steps were added: variable blur levels, histogram equalization, and noise reduction. These changes aim to improve the performance of the software in non-ideal conditions (at night, during rainy weather). In terms of features, the following were added: an option to separate each parking zone into multiple polygons to provide non-binary parking detection (multiple levels of parking occupancy depending on the percentage of the surface that is occupied), a variable edge detection threshold for each parking zone (instead of using a global one), and a second algorithm for detection which is luminance-based and detects if a given parking spot is free or occupied based on the level of light in a given area (since asphalt has a distinct luminance).

For traffic detection, the problem was divided in three distinct parts, which can be tested and improved separately, but must work together in order for the software to function: motion detection, object tracking, and object counting. Motion detection first involves separating objects in motion from the static background, using similar functions to those in [8], [9], and [10]. The following features were added: a shadow mask to solve the problem of objects being merged together due to shadow overlap, dilation to improve definition of small objects, and histogram equalization. Then, a feature that discards irrelevant contours (corresponding to non-pedestrian, non-vehicle objects) was implemented. The next step, object tracking, involves linking contours on one frame to ones on the previous, allowing tracking objects in time, not only space. The third step, object counting, is done as follows: an algorithm determines the direction of motion of objects that entered or exited the frame based on where new contours appeared and old contours disappeared, and counts how many objects have entered or exited from each border of the frame (left, top, right, and bottom). The overall results are shown graphically. A frame showing only objects in motion is displayed, and the detected contours are outlined. The color of the outline indicates the results of contour tracking: it changes gradually from one color to another based on how long a given object has been tracked.

2. Architecture and assumptions

2.1 Parking detection architecture

The structure of the parking detection software is shown in **Figure 1**. The user must supply a video input file, parking zone data (coordinates of parking zones), and a file containing various settings (for logic and image processing) (more details on these in section 3.1.1). The program performs image pre-processing on the video input to improve performance of the parking occupancy detection algorithms (one based on edges and one based on luminance). The outputs of these algorithms are then combined to produce the raw detection results, which are directly provided to the user as feedback (drawn on a video output frame and displayed). The results are also interpreted by an overall occupancy algorithm that filters out measurement noise and are then written to a database.

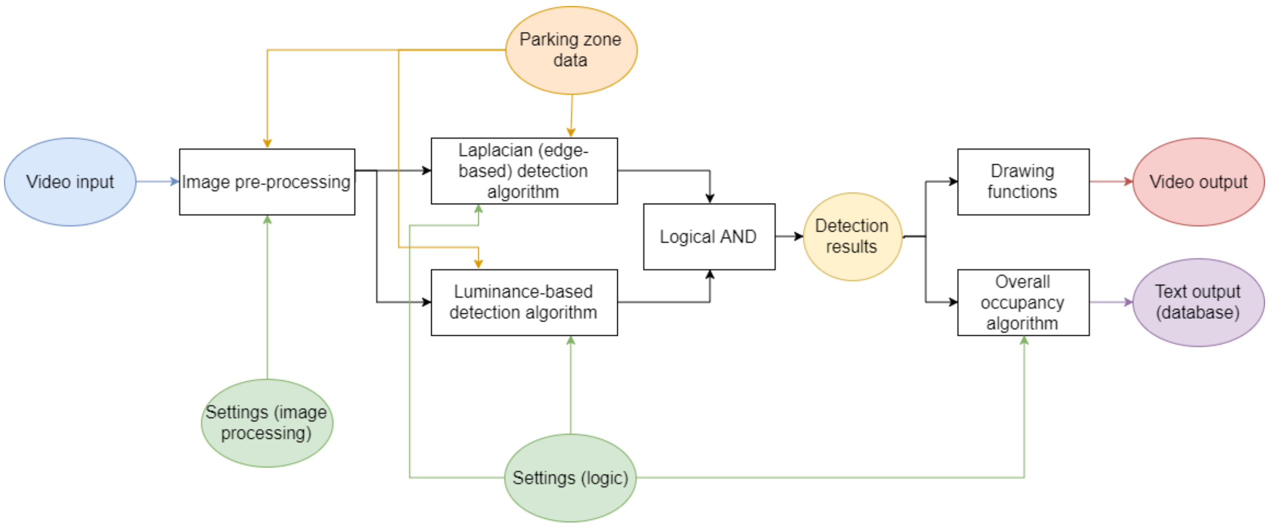


Figure 1: Parking data architecture block diagram

2.2 Parking detection assumptions

A few assumptions were made when developing this software, mainly regarding the user inputs. The video file must be of a certain level of sharpness for the edge-based detection algorithm to work properly: if it is too blurry, the edge detection algorithm will be unable to identify vehicle edges, and if it is too sharp, the algorithm could identify excessive contours on the asphalt (though this can be remedied with blur and noise reduction). Also, the camera inclination must be as low as possible: ideally, it would be pointed straight down, and software performance will degrade as it is tilted higher and higher (due to tall vehicles covering adjacent spots). The vacant parking spots should also be as clean as possible: any large debris, or marks or cracks on the pavement could be detected as edges and will reduce the probability of accurate detection. Finally, the luminance algorithm functions best with even lighting, so there are issues at night, when the sources of light (lamps) are scattered and lighting is uneven. For the parking zone data, the assumption is that the user has set the regions properly (see section 3.1.1 for more details): each region should be entirely covered by a polygon, and edges are specified in the correct order (clockwise, starting at top left). For the settings file, it is assumed that the user will provide rational values for each setting, following the guidelines and keeping every value within the same order of magnitude. Although there are checks for invalid values (such as negative numbers where a positive one is required), the program will run an irrational value, but provide nonsensical results.

2.3 Motion detection architecture

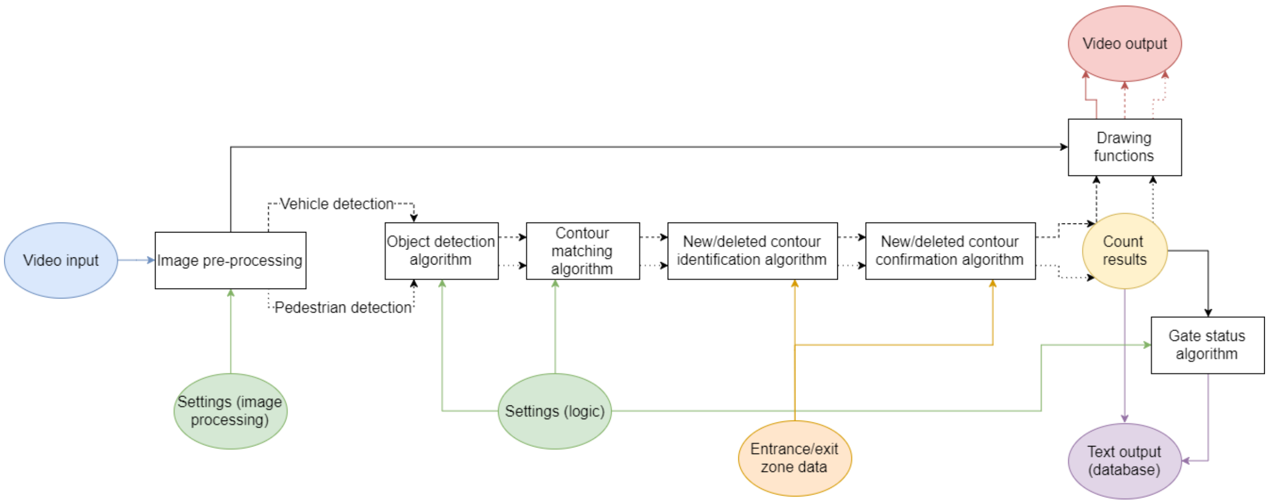


Figure 2: Motion detection architecture block diagram

The structure of the motion detection software is shown in **Figure 2** below. The user must supply a video input file, entrance/exit zone data (coordinates of entrance/exit zones), and the aforementioned settings file. The program performs image pre-processing on the video input to isolate moving objects and make them easier to identify. It outputs two different frames: one for vehicles and one for pedestrians, with different parameters for each, to optimize detection for each case. Then, an object detection algorithm detects contours of objects in the frame, and a contour matching algorithm allows tracking a given object by linking its contours between frames. For counting, a new/deleted contour identification algorithm identifies contours that have possibly entered or exited the frame and marks the entering/exiting locations, and a new/deleted contour confirmation algorithm confirms that the contours have indeed entered/exited the frame and generates the appropriate counts. The count results are combined with image pre-processing results and are displayed as video output to the user. They are also written to a database: some data is written immediately, other, periodically through a gate status algorithm.

2.4 Motion detection assumptions

For the motion detection algorithm, the requirements on the video file are the following. The camera inclination is critical: if it is tilted too high, or with the wrong perspective (for example, oblique instead of isometric on a multi-lane road), it will lead to a very significant error in the count, both in consistency and accuracy, as vehicles in one lane will obscure those in lanes behind them. For optimal camera placements, see section 5.2.1. Otherwise, scene lighting and sharpness are not as critical as with parking detection (see section 2.2) because the algorithm is based on motion, so changes in these parameters will not affect the difference frame. However, one key aspect of lighting that cannot be ignored is the presence of shadows: if they are large enough to cause nearby objects contours to overlap, precise tuning of the shadow mask (see section 3.4.2) will be needed to improve detection results, but they will never reach the consistency and accuracy of a scene without large shadows. For the entrance/exit zone data file, very precise tuning is not required, but a certain amount of experience is necessary for setting the boundaries properly. If they are not properly defined, it could lead to vehicles being missed or counted twice when they stop at an intersection, so some testing may be required to minimize the occurrence of these events. Finally, the motion detection settings are much trickier to define than the parking detection settings. The most difficult ones to calibrate are the perspective and shadow mask parameters, and although a visual feedback is provided to the user (see section 3.1.3), it is not trivial to determine which parameter should be tuned next. Even worse, the errors caused by an incorrect configuration of these parameters may be very subtle, so a decent understanding of the impact of each setting, as well as extensive testing, will be needed for optimal software performance.

3. Details of the solution

3.1 User interface

3.1.1 User input and configuration

The program requires a few launch parameters to function, which can be passed via the command line using the following syntax:

```
./smart-city <Video filename> <Parking data filename> <Gate data filename>
<Settings filename>
```

`<Video file name>` refers to the video stream to be analyzed, which can be either the path of a video file (recording) or a camera URL livestream). `<Parking data file name>` is a file containing information about the parking zones for parking detection. `<Gate data file name>` is a file containing information about the designated entrance and exit zones for motion detection. `<Settings filename>` corresponds to a file containing different numerical parameters that influence image processing or the functioning of parking or motion detection algorithms. Note that configuring the parking data file and gate data file requires finding coordinates of a point in the frame. There exists a

feature to simplify this task: the user must simply position the mouse at the desired point in the frame labeled “Video” and perform a left-click; the coordinates should appear in the console.
The parking data text file describes the parking zone, each on a separate line. The line contains an ID value (numerical value which will be used to label the parking spot on the output), followed by a type value (0 if the parking zone is a valid parking spot and 1 if it is an illegal zone to be monitored for violations). Then, there are the boundaries which are specified using a set of four (x,y) coordinates of the quadrilateral that defines the parking spot, where the (0,0) point corresponds to the upper left corner of the frame. The points must be specified in a clockwise order, starting from the top left corner. For optimal performance, the quadrilateral should cover the entire parking spot, but not parking spot markings such as lines. Finally, there is a threshold value for the minimum amount of detail that needs to be present in a parking spot for it to be considered as occupied. This value is unique to each parking spot and must be determined empirically. A value indicating the amount of detail detected in each parking polygon is provided in the console at each frame, so one must take note of different values when the parking spot is free and occupied, in different environmental conditions, then select a value that is higher than most or all recorded values when the spot is free, and lower than most or all recorded values when the spot is occupied. For example, the parking spot in **Figure 3** with coordinates at (420, 120), (470, 107), (442, 147), and (382, 163) and a threshold value of 5.0 would be specified by the following line:

```
0 1 420 120 470 107 442 147 382 163 5.0
```

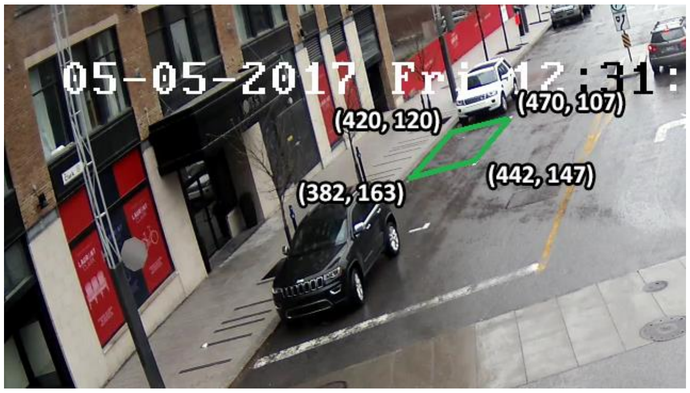


Figure 3: Parking spot defined by above line

The first part of the gate data text file, under the “Gates” label, describes each designated entrance and exit zone, each on a separate line, similarly to the parking data file. Each line starts with an ID value, but unlike the parking file, it is not arbitrary: “0” signifies “left”, “1” signifies “top”, “2” signifies “right”, and “3” signifies “bottom”. The next digit, the type, determines what the gate represents: “0” signifies a vehicle entrance, “1” signifies a vehicle exit, “2” signifies a pedestrian entrance, and “3” signifies a pedestrian exit. The next four numbers are the coordinates, but since gates are rectangular, only four numbers are needed, supplied in the following order: top corner x-coordinate, top-corner y-coordinate, width, height. For example, the left vehicle entrance in **Figure 4** with top left corner at (0, 60) and bottom right one at (400, 350) (length: 400, width: 290) is defined as:

```
0 0 0 60 400 290
```

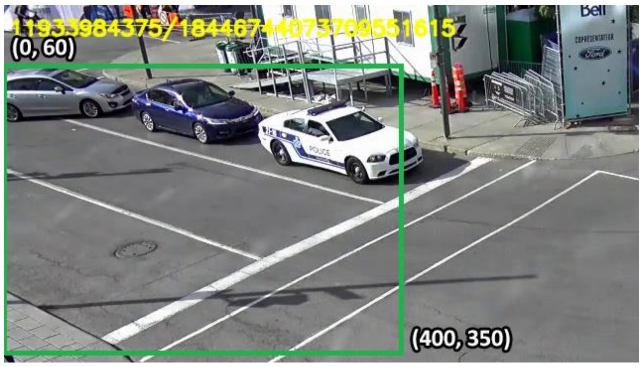
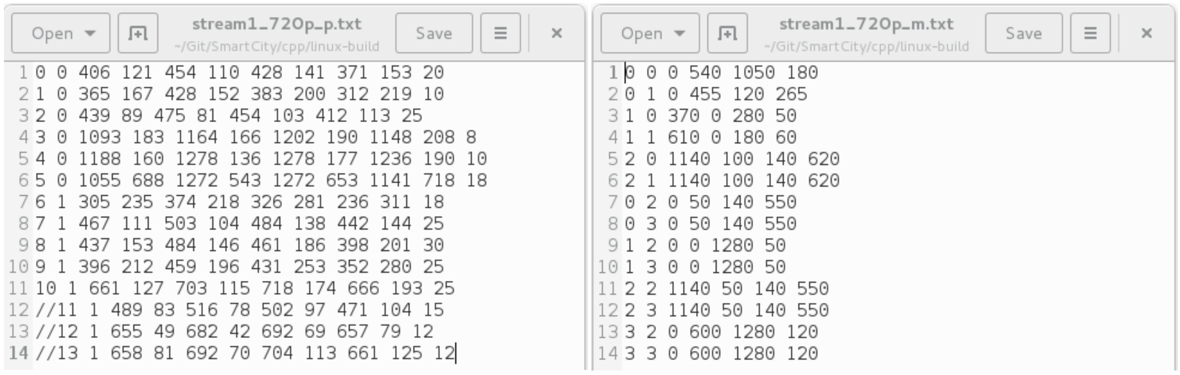


Figure 4: Vehicle entrance defined by above line

The next part of the gate data file, under the "Dead zones" label, allows the user to set dead zones where no motion detection can occur. The main purpose of dead zones is to be placed between gates and edges of the frame, to ensure that vehicle count still works with gates placed away from edges of the frame (for more details, see section 3.5.4). Deadzones are declared similarly to gates, but do not have an ID value (only the type, the top corner coordinates, and the dimensions).

The following image (**Figure 5**) shows examples of a parking data text file and a gate data text file:


Open
Save
stream1_720p_p.txt
~/Git/SmartCity/cpp/linux-build
stream1_720p_m.txt
~/Git/SmartCity/cpp/linux-build

```
0 0 406 121 454 110 428 141 371 153 20
1 0 365 167 428 152 383 200 312 219 10
2 0 439 89 475 81 454 103 412 113 25
3 0 1093 183 1164 166 1202 190 1148 208 8
4 0 1188 160 1278 136 1278 177 1236 190 10
5 0 1055 688 1272 543 1272 653 1141 718 18
6 1 305 235 374 218 326 281 236 311 18
7 1 467 111 503 104 484 138 442 144 25
8 1 437 153 484 146 461 186 398 201 30
9 1 396 212 459 196 431 253 352 280 25
10 1 661 127 703 115 718 174 666 193 25
//11 1 489 83 516 78 502 97 471 104 15
//12 1 655 49 682 42 692 69 657 79 12
//13 1 658 81 692 70 704 113 661 125 12
```

```
0 0 0 540 1050 180
0 1 0 455 120 265
1 0 370 0 280 50
1 1 610 0 180 60
2 0 1140 100 140 620
2 1 1140 100 140 620
0 2 0 50 140 550
0 3 0 50 140 550
1 2 0 0 1280 50
1 3 0 0 1280 50
2 2 1140 50 140 550
2 3 1140 50 140 550
3 2 0 600 1280 120
3 3 0 600 1280 120
```

Figure 5: Parking data text file (left) and gate data text file (right)

The "settings" file is another text file where every pair of lines is the description of a setting and, below, its value. Settings are parameters that are necessary to be adjustable by the user without having to obtain and modify the program's source code, so they are parsed from the file instead of being hardcoded. Some parameters are only for user convenience and do not affect the algorithm, such as the time interval between logging events to a file. Others, such as the number of polygons per parking spot, type of parking detection algorithm used, and perspective correction for object size do have a positive or negative effect on the reliability of the program, but it is immediately visible and easy to understand. Finally, most of the image processing parameters are for advanced users only. The default values are adequate, so modifying them is not critical, but an advanced user could tweak them to improve performance. The following table (**Table 1**) shows the categorized settings and their description:

Table 1: Parking and motion detection settings, categorized

Setting (section)	Description	Value range and typical value
Frame dimensions	Frame width and height in pixels	Integer value > 0 1280 720
Parking detection image processing settings		
Thickness of boundary of zone	How many pixels thick is the border of the area around each parking zone that is denoised,	Integer value > 0 10

for denoising (3.2.3)	higher: more accurate denoising but slower	
Blur level (3.2.1)	How much blur should be applied to the frame in pre-processing for parking detection, higher: more blur	Floating-point value >= 0 0.0
Denoising amount (3.2.3)	How strong the denoising operation should be, higher: more noise removal, lower: preserve more detail	Floating-point value > 0 4.5
Clipping limit for contrast adjustment for edge detection (3.2.2)	How intense histogram equalization can be for edge detection, higher: more contrast, lower: less noise	Floating-point value > 0 3.0
Clipping limit for contrast adjustment for luma detection (3.2.2)	How intense histogram equalization can be for luma detection, higher: more contrast, lower: less noise	Floating-point value > 0 2.5
Upper and lower limit of luminance of pavement (3.3.3)	Thresholds for detecting a vehicle based on luminance, upper limit: sensitivity to highlights, lower limit: sensitivity to shadows	255 >= Integer value >= 0 200 45
Parking detection logic parameters		
Number of regions per parking zone (3.3.1)	Number of regions in which each parking spot is split, higher: more possible intermediate occupancy states (1: 2, 2:3, or 4:5)	Either 1, 2, or 4 4
Parking detection algorithm (3.3.2, 3.3.3, 3.3.4)	Parking detection algorithm used (1: Laplacian, 2: luminance, 3: both)	Either 1, 2, or 3 3
Parking status update interval (3.3.1)	Time between each parking status update, higher: faster but more error-prone response, lower: slower but more accurate response	Floating-point value > 0 10
Opening and closing time of the parking lot	Times between which parking is globally disallowed (all spots marked as unavailable)	Hours in 24-hour format 9 21
Motion detection image processing parameters		
Blur level (3.4.1)	How much blur should be applied to the frame in pre-processing for motion detection, higher: more blur	Floating-point value >= 0 1.0
White threshold (large contours, top & bottom part) (3.4.3)	Threshold for difference frame pixels to be cast to white in thresholding operation for large contours (vehicles), lower: more sensitive to motion, different values for top & bottom part of frame	255 >= Integer value >=0 6 8
Dilation kernel size (large contours, bottom & top part) (3.4.3)	Size of dilation kernel for large contours (vehicles), higher: more chance of grouping contours together, different values for top & bottom part of frame	Integer value >= 1 7 11
White threshold (small contours) (3.4.3)	Threshold for difference frame pixels to be cast to white in thresholding operation for small contours (pedestrians), lower: more sensitive to	255 >= Integer value >= 0 10

	motion	
Dilation kernel size (small contours) (3.4.3)	Size of dilation kernel for small contours (pedestrians), higher: more chance of grouping contours together	Integer value >= 1 15
Shadow mask threshold for hue (3.4.2)	Upper threshold for angular difference between foreground and background hue to consider foreground as shadow	180 >= Integer value >= 0 70
Shadow mask thresholds for saturation (3.4.2)	Upper and lower thresholds for absolute difference between foreground and background saturation to consider foreground as shadow	255 >= Integer value >= 0 70 10
Shadow mask thresholds for value (3.4.2)	Upper and lower thresholds for ratio between foreground and background value to consider foreground as shadow	1>= Floating-point value >= 0 0.5 0.12
Vertical coordinate for top/bottom frame separation (3.4.3)	Vertical coordinate separating the top and bottom of frame for different image processing values above	Integer value >= 0 240
Motion detection logic parameters		
Gain for area calculations (large contours) (3.5.2)	Scaling factor for minimum size contour that is considered as a vehicle, considering perspective, higher: less tolerance for small contours	Floating-point value > 0 0.02
Gain for distance calculations (large contours) (3.5.2)	Scaling factor for maximum distance between vehicle contour centroids to link them between frames, considering perspective, higher: more tolerance for fast-moving contours	Floating-point value > 0 0.2
Gain for area calculations (small contours) (3.5.2)	Scaling factor for minimum and maximum size contour that is considered as a pedestrian, considering perspective, higher: less tolerance for small contours, more for large ones	Floating-point value > 0 0.005
Gain for distance calculations (small contours) (3.5.2)	Scaling factor for maximum distance between pedestrian contour centroids to link them between frames, considering perspective, higher: more tolerance for fast-moving contours	Floating-point value > 0 0.02
Horizontal and vertical perspective multipliers (3.5.2)	Multipliers applied to horizontal and vertical distances from the origin, higher: increase effect of relevant dimension for perspective compensation	Floating-point value > 0 0.25 1.45
Horizontal and vertical perspective origin multipliers (3.5.2)	Fractions of frame width and height where origin located, higher: origin moves towards right/bottom (respectively)	1 >= Floating-point value >= 0 0.3 1
Offset for perspective (3.5.2)	Value of perspective correction calculation for object located exactly at camera origin	Integer value, minimum: distance between origin and farthest point on frame, taking into account multipliers 1600 (for 1280x720)
Traffic status update	Time between each traffic status update, higher:	Floating-point value > 0

interval (3.5.3.3.3)	faster updates	10

3.1.2 Parking detection program output

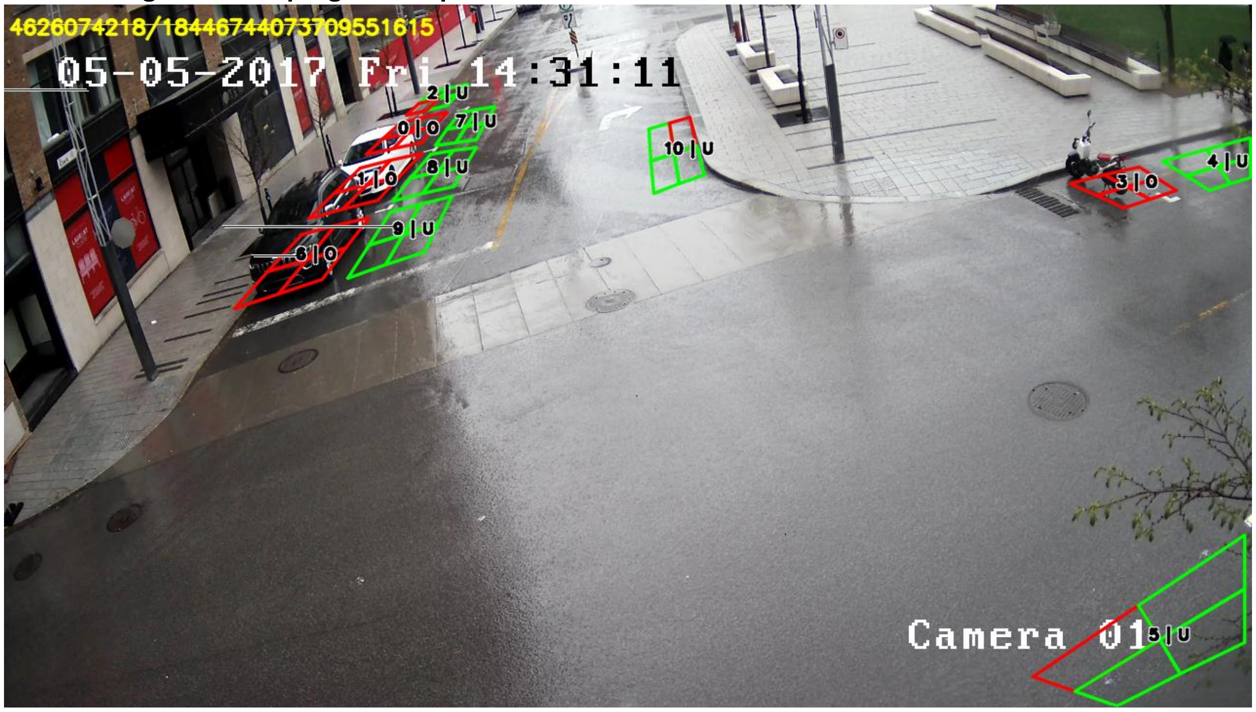
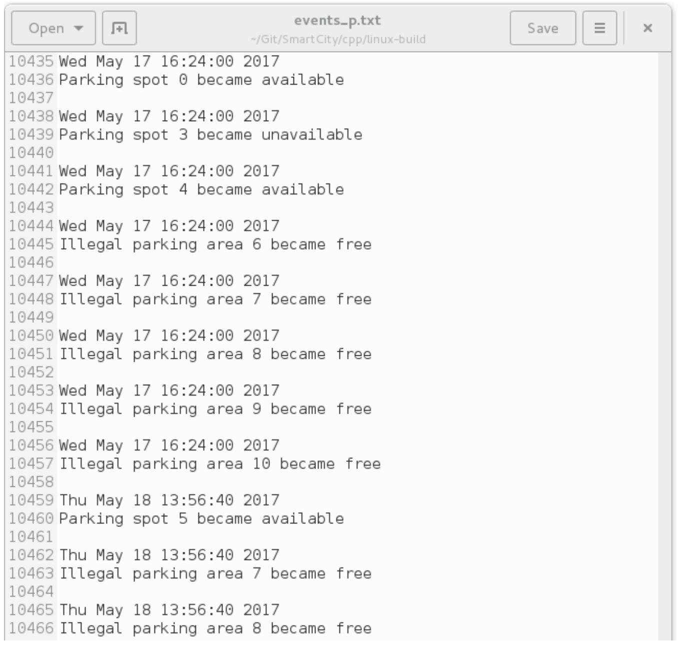


Figure 6: Parking detection user interface (output frame)



```
Wed May 17 16:24:00 2017
Parking spot 0 became available

Wed May 17 16:24:00 2017
Parking spot 3 became unavailable

Wed May 17 16:24:00 2017
Parking spot 4 became available

Wed May 17 16:24:00 2017
Illegal parking area 6 became free

Wed May 17 16:24:00 2017
Illegal parking area 7 became free

Wed May 17 16:24:00 2017
Illegal parking area 8 became free

Wed May 17 16:24:00 2017
Illegal parking area 9 became free

Wed May 17 16:24:00 2017
Illegal parking area 10 became free

Thu May 18 13:56:40 2017
Parking spot 5 became available

Thu May 18 13:56:40 2017
Illegal parking area 7 became free

Thu May 18 13:56:40 2017
Illegal parking area 8 became free
```

Figure 7: Parking detection log file

The parking detection software presents the user with a window displaying the current frame of the stream being analyzed, with an overlay of the defined parking zones and their associated ID in the center. They are split into 1, 2, or 4 sub-zones depending on user configuration (see section 3.3.1). Each individual sub-zone is highlighted green if detected as free and red if occupied (this status updates in real-time and provides visual feedback). Each zone also has a letter near its ID value that corresponds to the overall zone's occupancy status: "U" for unoccupied, and "O" for occupied (this status updates at regular

time intervals to avoid multiple toggles causing noise). **Figure 6** demonstrates the output of the parking detection software.
The parking detection program also logs any changes in parking state of a given zone to a text file, along with a timestamp (time & date). This is a precursor to writing a full function that writes to a database. A parking zone will only be considered as available when its occupancy reaches 75% or above, and unavailable when the percentage drops to 25% or below. The type of the parking zone also influences the message that will be logged in the database: for a parking spot, it would be "Parking spot n became available/unavailable", and for an illegal zone, it would be "Illegal parking area became free/blocked", depending the value of occupancy. Figure 7 shows an example of the parking data log:
3.1.3 Motion detection program output

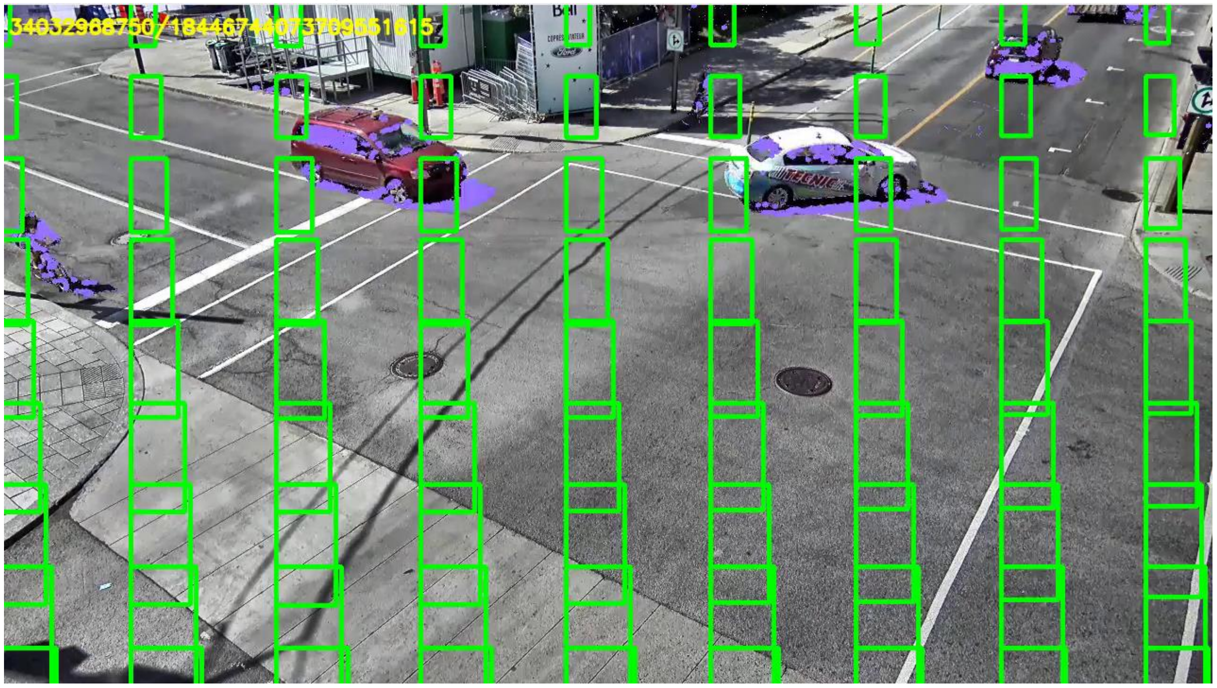


Figure 8: Motion detection user interface (original frame, shadow mask and perspective correction)

The motion detection program presents the user with two types of windows: the original frame, and difference frames (one for vehicle and one for pedestrians). The original frame is almost identical to the frame captured by the camera, but includes an overlay of the shadow mask in purple (see section 3.4.2), and an overlay of the perspective verification rectangles in green (see section 3.5.2). The above features are only for verifying the validity of shadow mask threshold and perspective compensation parameters and can be disabled in the settings. The difference frame shows the program's interpretation of the video stream and provides visual feedback of detected and identified objects to the user so they can tweak settings until the program performs optimally. There is one difference frame for vehicles and one for pedestrians to avoid excessive clutter. Each final difference frame is generated by taking the original difference frame (after dilation operations, see section 3.4.3), and overlaying the contours of detected objects (derived from the threshold frame, see section 3.4.3), as well as the boundaries of the designated entrance/exit zones and dead zones (see section 3.1.1). The object contours outlines are colored using a color gradient (from red to yellow to green) depending on the lifetime of the contour (vehicles: red for a lifetime of 0 to green for a lifetime of 5+, pedestrians: red for a life time of 0 to green for a lifetime of 13+). Furthermore, the ID of each contour is to help the user determine if a given object is being tracked properly across the frame. As for the designated zones, the entrance zones are yellow, the exit zones are blue, and the combined entrance/exit zones are white. The following images show examples of an original frame with the shadow mask and perspective correction enabled (**Figure 8**), and a difference frame (**Figure 9**) from the same moment:

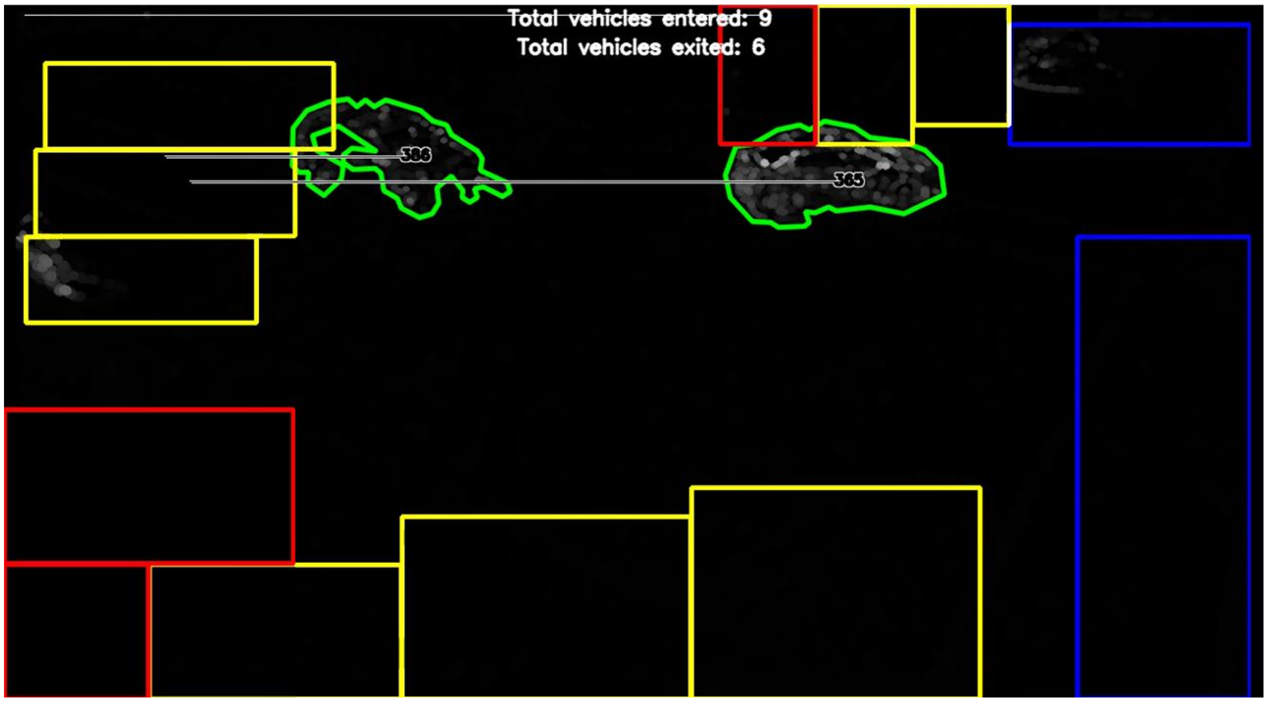
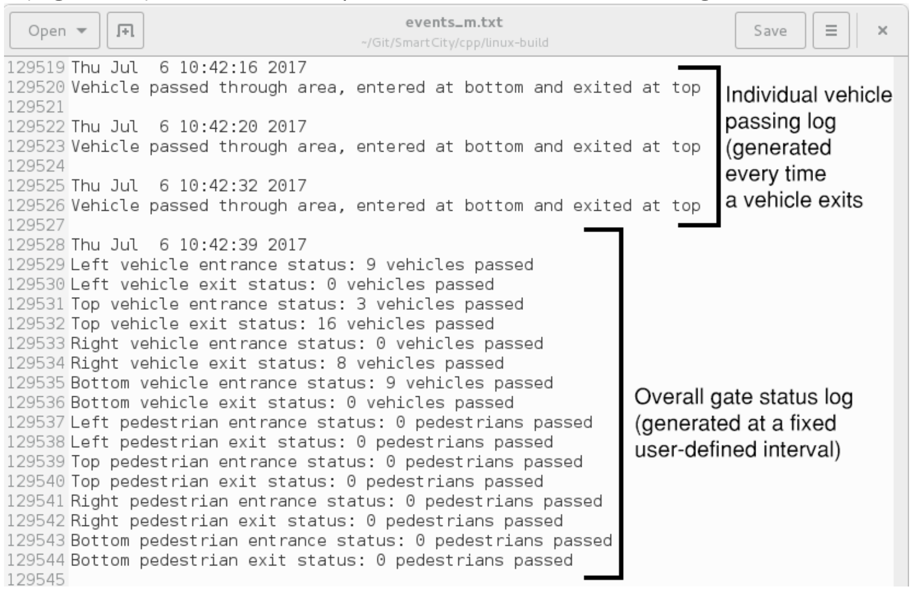


Figure 9: Motion detection user interface (difference frame showing detected vehicles and designated entrance/exit zones, as well as dead zones.)

The motion detection software also records two types of events to a log file, a precursor to writing this data to a database. Any time a pedestrian or vehicle exits the frame, its point of origin (if possible) and point of exit are recorded, in order to track not only how many vehicles enter and exit, but also the trajectory they follow. Furthermore, at regular, user-defined intervals, the status of all designated entrance/exit zones (i.e. number of vehicles that passed through each) is written to a log file. The following image (**Figure 10**) shows an example of the motion detection log file:


Open
events_m.txt
~/Git/SmartCity/cpp/linux-build
Save
Individual vehicle passing log (generated every time a vehicle exits
Overall gate status log (generated at a fixed user-defined interval)

```
Thu Jul  6 10:42:16 2017
Vehicle passed through area, entered at bottom and exited at top

Thu Jul  6 10:42:20 2017
Vehicle passed through area, entered at bottom and exited at top

Thu Jul  6 10:42:32 2017
Vehicle passed through area, entered at bottom and exited at top

Thu Jul  6 10:42:39 2017
Left vehicle entrance status: 9 vehicles passed
Left vehicle exit status: 0 vehicles passed
Top vehicle entrance status: 3 vehicles passed
Top vehicle exit status: 16 vehicles passed
Right vehicle entrance status: 0 vehicles passed
Right vehicle exit status: 8 vehicles passed
Bottom vehicle entrance status: 9 vehicles passed
Bottom vehicle exit status: 0 vehicles passed
Left pedestrian entrance status: 0 pedestrians passed
Left pedestrian exit status: 0 pedestrians passed
Top pedestrian entrance status: 0 pedestrians passed
Top pedestrian exit status: 0 pedestrians passed
Right pedestrian entrance status: 0 pedestrians passed
Right pedestrian exit status: 0 pedestrians passed
Bottom pedestrian entrance status: 0 pedestrians passed
Bottom pedestrian exit status: 0 pedestrians passed
```

Figure 10: Motion detection log file

3.2 Parking detection image processing

The parking detection image processing pipeline algorithm is shown in **Figure 11** below. It consists of three main steps: grayscale conversion/blur, histogram equalization, and denoising (noise reduction), which are outlined in sections 3.2.1, 3.2.2, and 3.1.3 below.

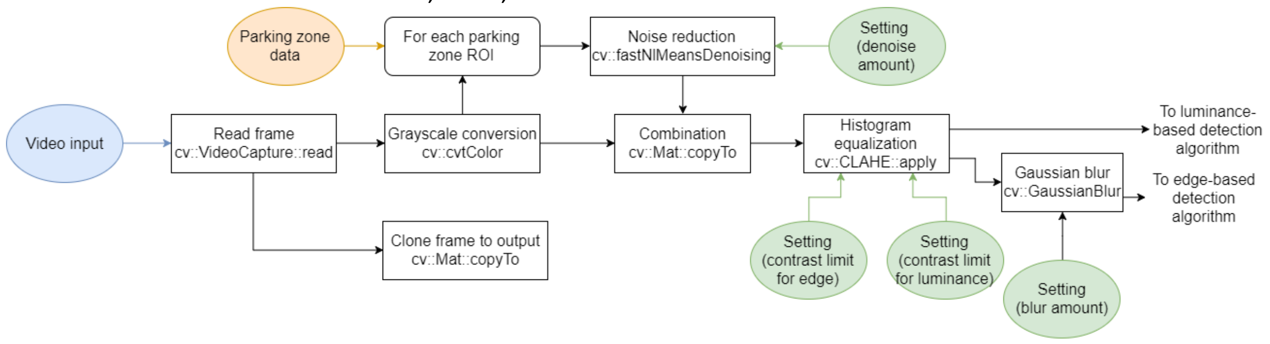

Parking zone data
For each parking zone ROI
Noise reduction cv::fastNlMeansDenoising
Setting (denoise amount)
Video input
Read frame cv::VideoCapture::read
Grayscale conversion cv::cvtColor
Combination cv::Mat::copyTo
Histogram equalization cv::CLAHE::apply
To luminance-based detection algorithm
Gaussian blur cv::GaussianBlur
To edge-based detection algorithm
Clone frame to output cv::Mat::copyTo
Setting (contrast limit for edge)
Setting (contrast limit for luminance)
Setting (blur amount)

Figure 11: Parking detection image processing pipeline

3.2.1 Grayscale conversion and blur

The detection algorithm is based on edge detection. The premise is that an empty parking zone will have fairly uniform color due to the asphalt surface, while an occupied one will feature more edges due to the various features of the vehicle that occupies it. During each program loop, a frame is captured. The frame must undergo some pre-processing before it is ready to be analyzed. First, the frame is converted from color to grayscale using *Formula 1*, which determines each pixel's luminosity based on the red, green, and blue channels [11]:

$$Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \qquad (1)$$

After that, the image is blurred to eliminate excessive details that would manifest themselves as noise in the algorithm. The Gaussian blur function performs the convolution of the original image with the specified kernel [12], which was chosen to be a 5x5 rectangle so that the value of each blurred pixel depends on a sufficient, but not excessive number of nearby samples. The other parameters are the standard deviation values (sigmaX and sigmaY), which are used to compute the weights of the other pixels using a Gaussian distribution curve. In other words, each pixel's value is replaced with a weighted average of the surrounding 5x5 grid of pixels, with the weight of each pixel being lower the farther it is from the original one. The standard deviation can be configured by the user in the settings file (the lower it is, the less blur, and a value of 0 disables blur completely). Generally, the best detection is achieved at a blur level of 0, but if the image is very sharp and there are excessive details detected on asphalt, the blur level can be increased.

3.2.2 Histogram equalization

In order to make the luminance-based detection algorithm function at night, it was necessary to make the brightness and contrast of the image uniform regardless of ambient light levels. This is achieved through histogram equalization. This procedure takes the histogram of the brightness of pixels in the image (a representation of how many pixels there are at each brightness level) and transforms it so that it occupies the whole brightness spectrum (i.e. the darkest pixels have the lowest possible brightness value for the specified color space and the brightest ones, the highest possible value) [13]. The specific implementation of histogram equalization used is the contrast limited adaptive histogram equalization, called `cv::CLAHE` in OpenCV [13]. This technique is superior to simple histogram equalization due to its adaptive nature: it only performs histogram equalization over small blocks of pixels (8x8 by default) [13]. This should solve the problem of bright areas becoming overly bright and dark areas becoming overly dark, which happens in ordinary histogram equalization where the contrast of the whole image is considered and results in loss of detail [13]. It also supports contrast limiting, which clips the contrast adjustment to a maximum specified clip limit and redistributes any excess uniformly across the histogram (i.e. to all pixels) [14]. For a more technical description of histogram equalization with contrast limiting, see [14]. This feature was used to specify different contrast limits for the edge-based and luminance-based algorithms so that each performs optimally. The benefits of contrast limited adaptive histogram equalization can be seen from the following images:

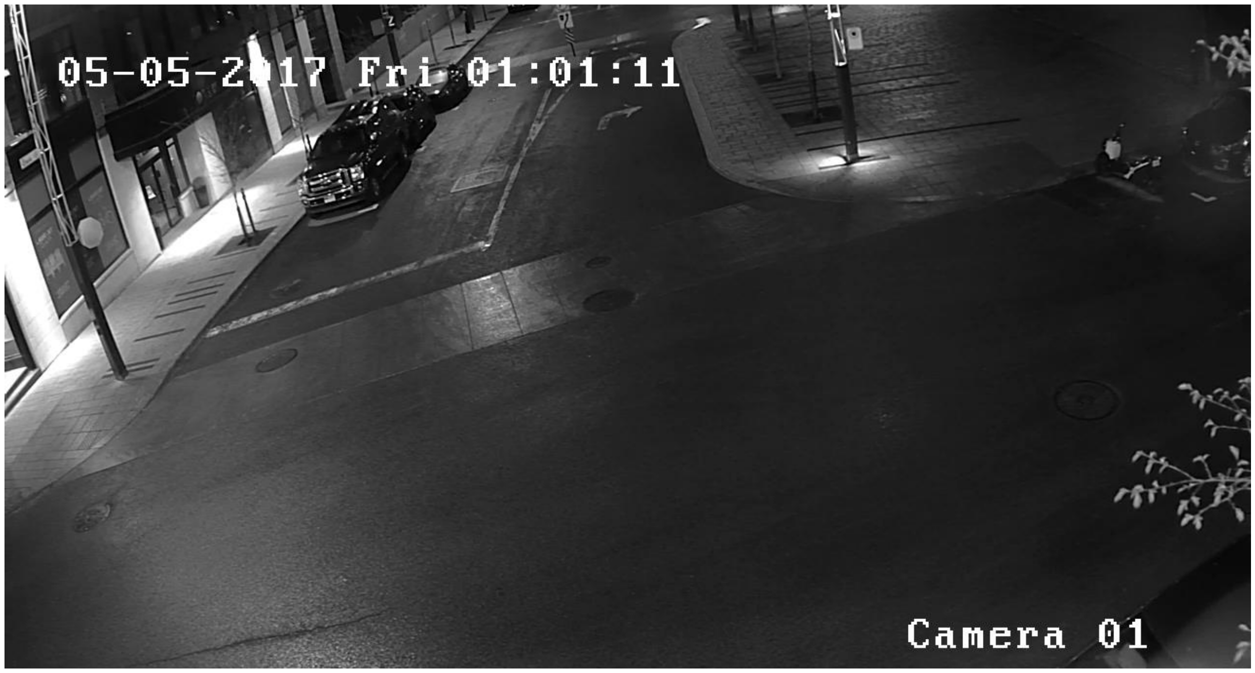


Figure 12: Night view without contrast-limited adaptive histogram equalization

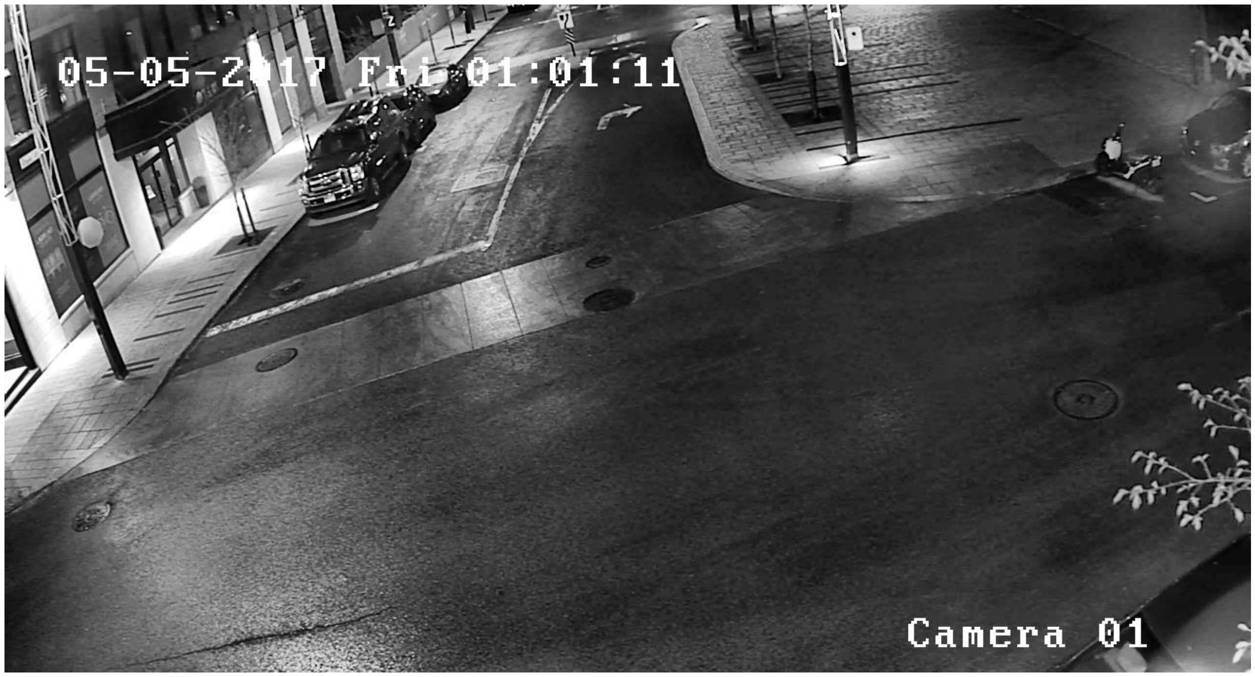


Figure 13: Night view with contrast-limited adaptive histogram equalization

In Figure 12, where CLAHE was not applied, areas of the image that are not under direct illumination by a street lamp are very dark and uniform, making it very difficult to detect edges. Even a person may have difficulty determining the state of a parking spot at a glance, so the task is almost impossible for the type of algorithm in use in this project. However, in Figure 13, the contrast of the image is significantly improved, almost to the point of resembling a daytime image.

3.2.3 Denoising

Another instance where obtaining reliable parking detection was difficult was during or after rain. Indeed, the image texture of the asphalt changes significantly when it is wet (the number of edges detected increases), leading to difficulties with the use of a fixed edge threshold parameter. A possible solution is blurring the image, but this technique affects relevant details in the image and makes detection more difficult. Instead, since the texture of the wet asphalt resembles random image noise, a noise reduction algorithm would be more appropriate. The fast non-local means denoising algorithm provided by OpenCV was used [15]. This algorithm analyzes the image to find regions that are similar, then replaces the pixels in each region by the average of the pixels of all the region, which works particularly well because noise is typically random [15]. For a more technical description of this algorithm, see [16]. The amount of denoising applied is controlled by a parameter in the settings file.

Unfortunately, this operation is very slow. Before applying this operation, the frame processing time (average during a short clip) was approximately 86.02 ms (leading to a frame rate of 11.63 FPS), which is more than sufficient for this application). Once denoising was applied, the frame processing time increased to 382.53 ms (2.61 FPS). Although this does not impact the parking detection application (which is not extremely time-sensitive), it is very unpleasant to watch and could preclude future applications such as counting that require a higher frame rate. Therefore, the algorithm was optimized to only affecting parking zones (and a small user-defined exterior border around them). After applying this optimization, the frame processing time was reduced to 197.58 ms (5.06 FPS). When considering only the time taken for the denoising algorithm (296.51 ms without optimization, 111.56 ms with optimization), this is a 2.66-fold improvement. The following images show the effect of the denoising operation, as well as an example of the other solution, blur. The effect has been exaggerated for clarity, but the principle is the same.

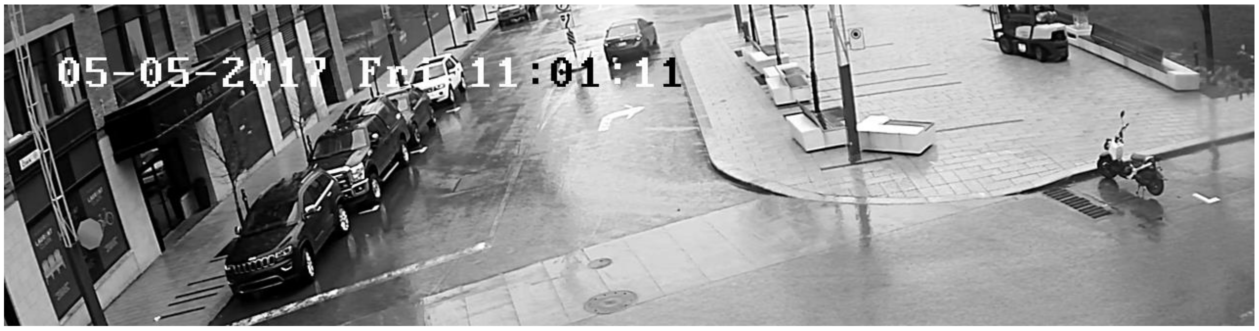


Figure 14: No noise removal effect

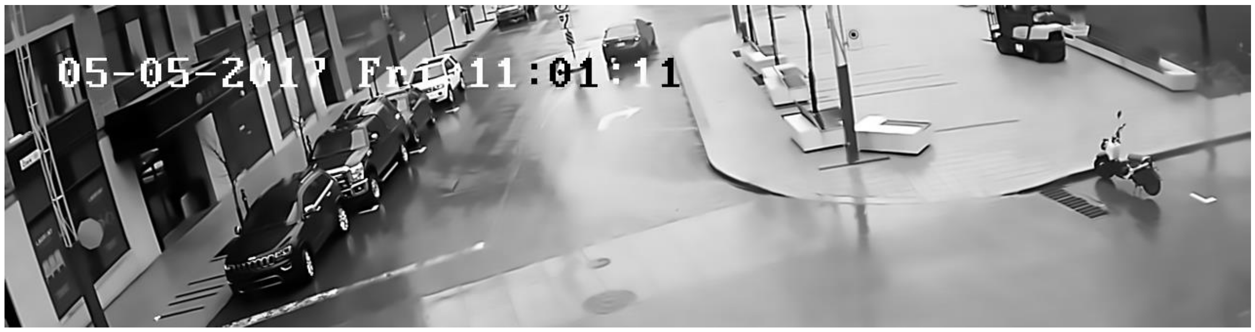


Figure 15: Fast non-local means denoising applied to whole image

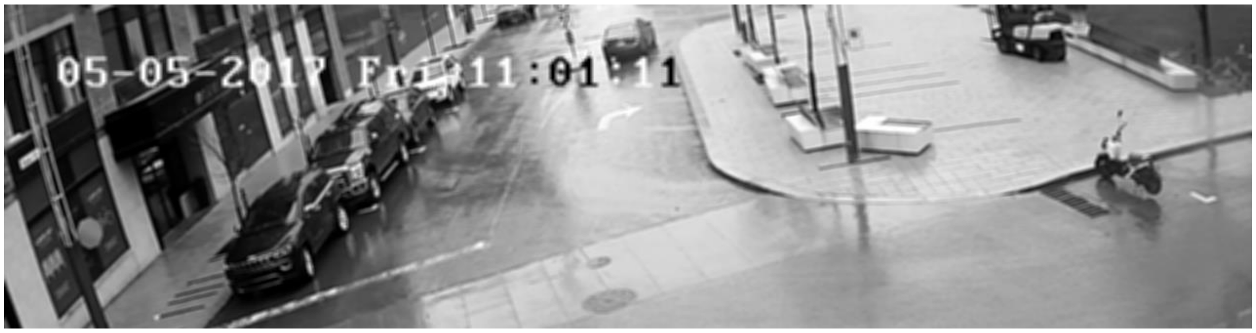


Figure 16: Blur applied to whole image

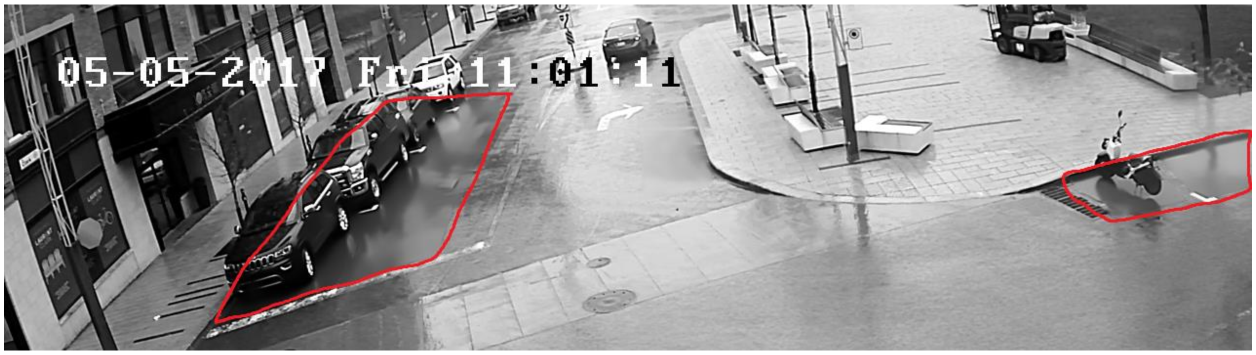


Figure 17: Fast non-local means denoising applied to part of image (circled areas)

Figure 14 shows the problem that had to be solved: the texture of the wet asphalt when it rains does not appear smooth on the camera feed.

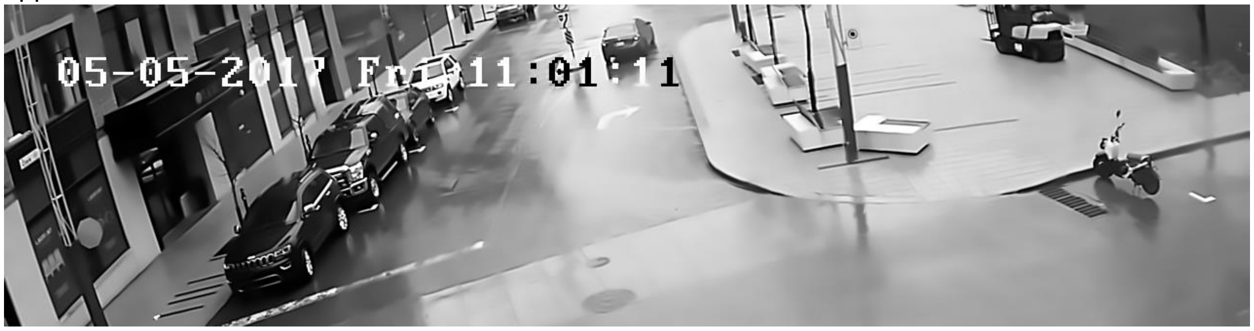


Figure 15 shows the application of the denoising algorithm: even though the texture of the asphalt has been significantly smoothed, the details on the surrounding vehicles are still clearly visible. Figure 16 shows the application of blur, which produces a poor effect: it is not as effective at removing the details on the asphalt, but a significant part of the details on the vehicles is lost. Finally,

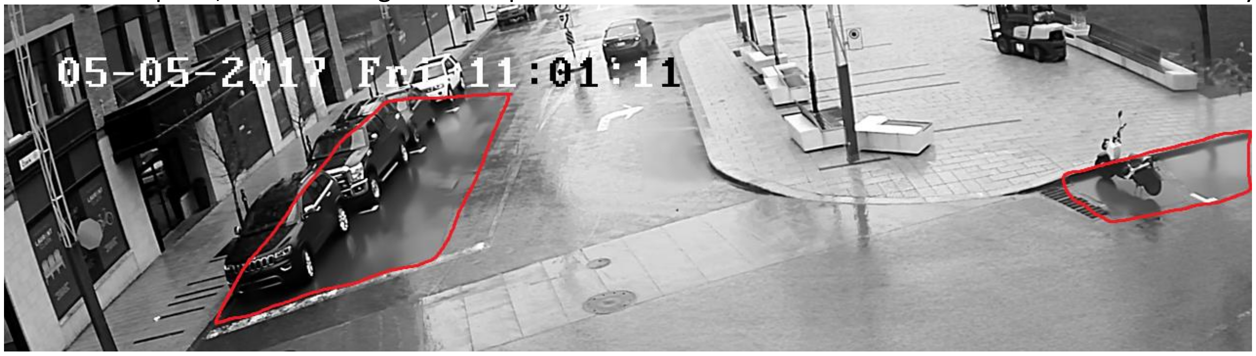


Figure 17shows the results of applying denoising only on the regions of interest.

3.3 Parking detection features and algorithms

3.3.1 Parking polygons and groups

The main challenge parking detection poses concerns the camera's perspective view of the parking spots, which makes it difficult to determine the state of a parking spot. Indeed, part of the car will block the parking spot behind it, and it is also marked as occupied, whereas it is in fact free. The solution that was implemented allows the user to define multiple polygons per parking spot: 1, 2, or 4. This was done by modifying the zone initialization function found in utils.cpp. The main change was assigning 1, 2, or 4 polygons per parking spot defined by the user, by partitioning the area down the middle horizontally or vertically wherever appropriate.

Each polygon is treated individually by the parking algorithm and assigned a value of 1 (occupied) or 0 (free). However, the end goal is to obtain a single occupancy state for a given parking spot (occupied or unoccupied) that can be provided to the user. This is achieved by introducing a new class ParkingGroup which groups the polygons that belong to a given parking spot (it essentially holds a vector of parking polygons).

The ParkingGroup class features the following methods:

- void addParking(Parking &park) to add parking polygons to a given parking spot
- Parking *getParking(int i) to retrieve a particular parking polygons (specifically a pointer to it)
- int getSize(void) to retrieve the number of parking polygons in a parking spot
- vector<cv::Point> getPoints(void) to retrieve the coordinates of the parking spot itself
- cv::Point getCenterPoint(void) to retrieve the coordinates of the center point of the parking spot itself
- int getOccupancy(void) to determine the occupancy of a parking spot as a percentage of parking polygons that are free

For one polygon, the occupancy percentage is still a binary case (either 0% or 100%), but two polygons allows 0%, 50%, or 100% occupancy, and four polygons is the best with 0%, 25%, 50%, 75%, or 100% occupancy. The main advantage of this method is that it allows filtering out actual changes in parking from simple noise in the reported delta value that causes a parking spot to change state momentarily. The actual status of a given parking spot is only checked when a user-defined number of seconds has passed.

For a comparison of the results obtained with 1, 2, and 4 zones, see the image below:

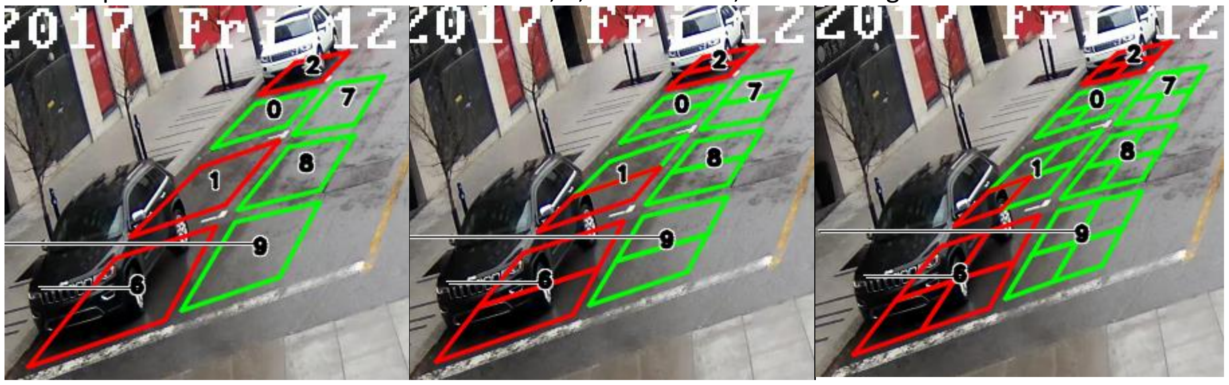


Figure 18: Comparison of parking detection performance with 1, 2, and 4 zones

From Figure 18 above, it is evident that the single polygon per parking zone approach is not optimal in this scenario, and that two and four polygons provide a much better result. Indeed, consider parking zone 1, which is not occupied, but partially covered by the large vehicle in zone 6. The one polygon approach marks it as occupied, which is completely wrong. The two polygon marks it as 50% free (1/2 free polygons), an indeterminate state, which is preferable to making the wrong decision. Finally, the four polygon approach marks it as 75% free (3/4 free polygons), which can be interpreted as free, since the occupancy percentage is below 50%.

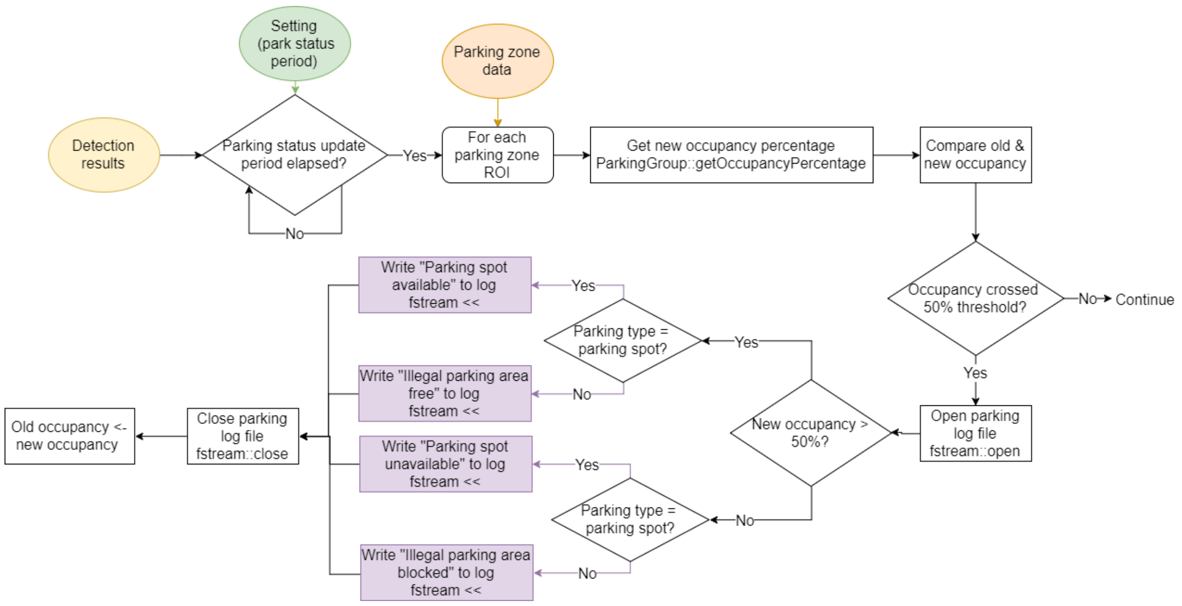

Setting (park status period)
Parking zone data
Detection results
Parking status update period elapsed?
Yes
No
For each parking zone ROI
Get new occupancy percentage ParkingGroup::getOccupancyPercentage
Compare old & new occupancy
Occupancy crossed 50% threshold?
No
Continue
Yes
Open parking log file fstream::open
New occupancy > 50%?
Parking type = parking spot?
Write "Parking spot available" to log fstream <<
Write "Illegal parking area free" to log fstream <<
Write "Parking spot unavailable" to log fstream <<
Write "Illegal parking area blocked" to log fstream <<
Close parking log file fstream::close
Old occupancy <- new occupancy

Figure 19: Parking detection overall occupancy algorithm

Furthermore, the fact that a parking spot can have up to 5 states (with four polygons) allows a more accurate checking of transitions. Indeed, an algorithm was devised where a parking spot will only transition to "free" when its occupancy percentage exceeds 50% (more than half the polygons are free), and "occupied" when its occupancy percentage falls below 50% (less than half the polygons are free). If the percentage hits exactly 50% (half the polygons), a change of state will not occur. This is a very noise-robust and accurate method for checking of parking spot status (it inspired by the concept of rising and falling edges in digital signals). This algorithm is the basis for writing parking status to the database, as it is advantageous to remove frequent toggling due to noise prior to logging events. It is outlined in **Figure 19**.

3.3.2 Laplacian algorithm

After the image processing techniques (see section 3.2) have been applied, the frame is ready to be analyzed for edge detection. This is done mainly through the Laplacian operator (OpenCV function `cv::Laplacian`). The overall algorithm is shown the following image (**Figure 20**). For each parking spot, the part of the frame bounded by the parking zone quadrilateral is isolated, generating a region of interest. Then, a Laplacian operator is applied to the ROI. This operator produces an image of the same size as the original one, where each pixel's value is the double derivative of the original pixel (in both dimensions since images are 2D): $\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$ [17]. This implements edge detection because on edge transitions, the concavity of the original function of pixels changes, therefore there is a maximum/minimum in the first derivative, and a zero in the second derivative [17]. The overall result is an image where the uniform surfaces are dark and the edges (transitions between surfaces) are bright, as seen in the example in [17]. The mean value of the pixels in the resulting region is then calculated, yielding a number that is higher the more edges there are in the image. Then, for each parking spot, the mean value is compared to the detection threshold and the result is assigned to a status variable for each parking spot, which is 0 if the average is bigger or equal to the threshold (parking spot occupied) and 1 if it is below the threshold (parking spot free). There is a unique threshold value for each parking spot instead of a global value because that would only be adequate if all parking spots were more or less uniform. However, in this case, the condition of the road and, more importantly, the typical orientation of a vehicle are different from spot to spot, which leads to a vast difference in reported Laplacian value between spots even if they have the same state. Adding a different threshold for a given parking spot is the ideal solution for this issue. Although this seems less user-friendly, as the user has to enter a specific value for each spot, it is in fact less tedious than trying to find a global one-size-fits-all value in most cases. In fact, detection will be reasonably accurate when using the same value for most spots, and setting a different one for a few problematic ones, so the amount of set up time is not excessive. **Figure 21** shows the performance of the Laplacian algorithm.

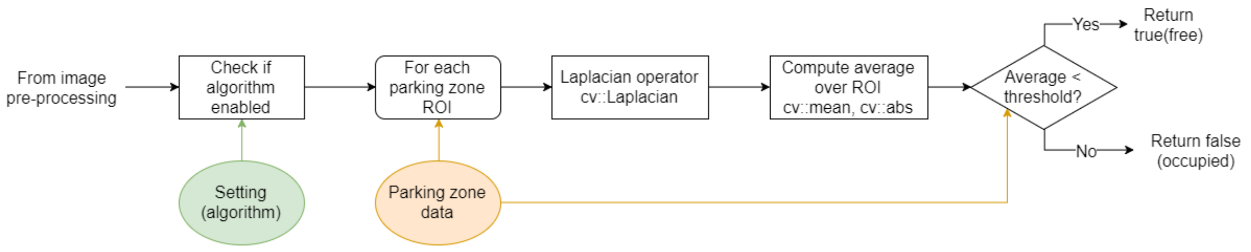

From image pre-processing
Check if algorithm enabled
Setting (algorithm)
For each parking zone ROI
Parking zone data
Laplacian operator cv::Laplacian
Compute average over ROI cv::mean, cv::abs
Average < threshold?
Yes
Return true(free)
No
Return false (occupied)

Figure 20: Laplacian detection algorithm

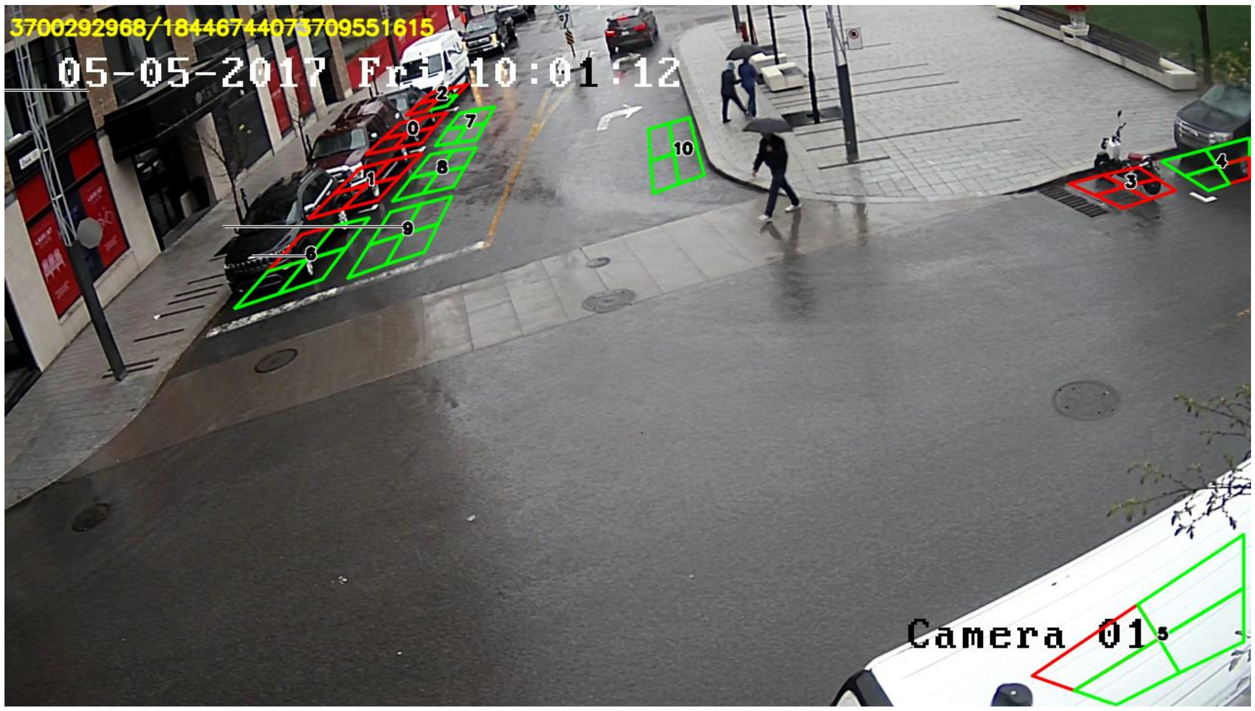


Figure 21: Performance of edge-based detection algorithm only

3.3.3 Luminance-based algorithm

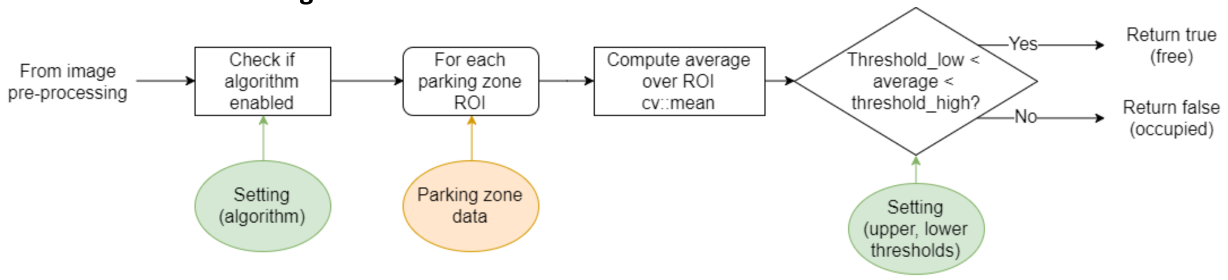

From image pre-processing
Check if algorithm enabled
Setting (algorithm)
For each parking zone ROI
Parking zone data
Compute average over ROI cv::mean
Threshold_low < average < threshold_high?
Setting (upper, lower thresholds)
Yes
Return true (free)
No
Return false (occupied)

Figure 22: Luminance detection algorithm

The next main challenge was improving parking detection in non-ideal conditions. The edge-based algorithm still suffers at detecting objects in regions where there are a lot of highlights and shadows, due to the relative uniformity of pixels in these areas. Therefore, a second algorithm, which is based on luminance, was implemented. It is shown in **Figure 22** below. The algorithm works by calculating the average luma (Y) value in a given region, which is very simple because the image is converted to grayscale, so the Y value is available explicitly (see section 3.2.1). This value is then compared to an upper threshold and lower threshold, and if it is above or below the nominal range of values, the spot is marked as occupied. This works because asphalt is grey, so it is roughly in the middle of the luminance spectrum. A very bright region corresponds to a white/pale vehicle occupying a spot, and a very dark one indicates a black vehicle or the shadow under the wheels of a vehicle. **Figure 23** shows the performance of the luminance algorithm.

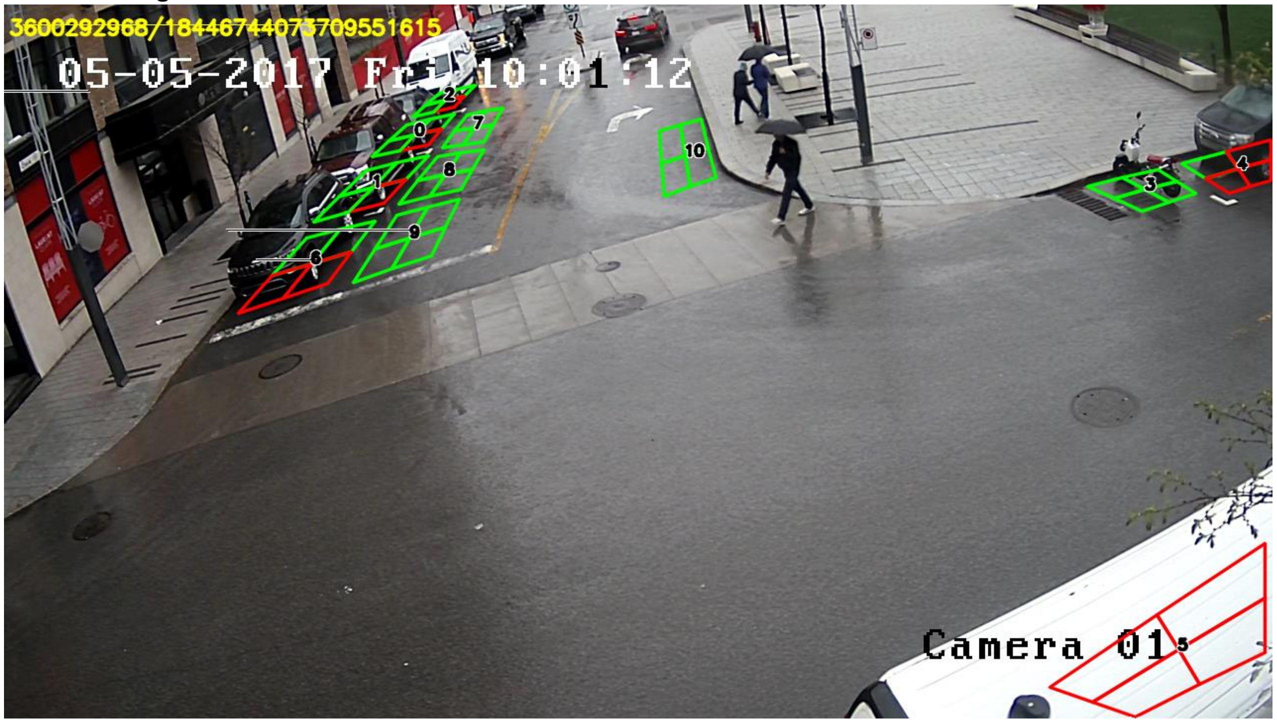


Figure 23: Performance of luminance-based detection algorithm only

3.3.4 Combination of algorithms

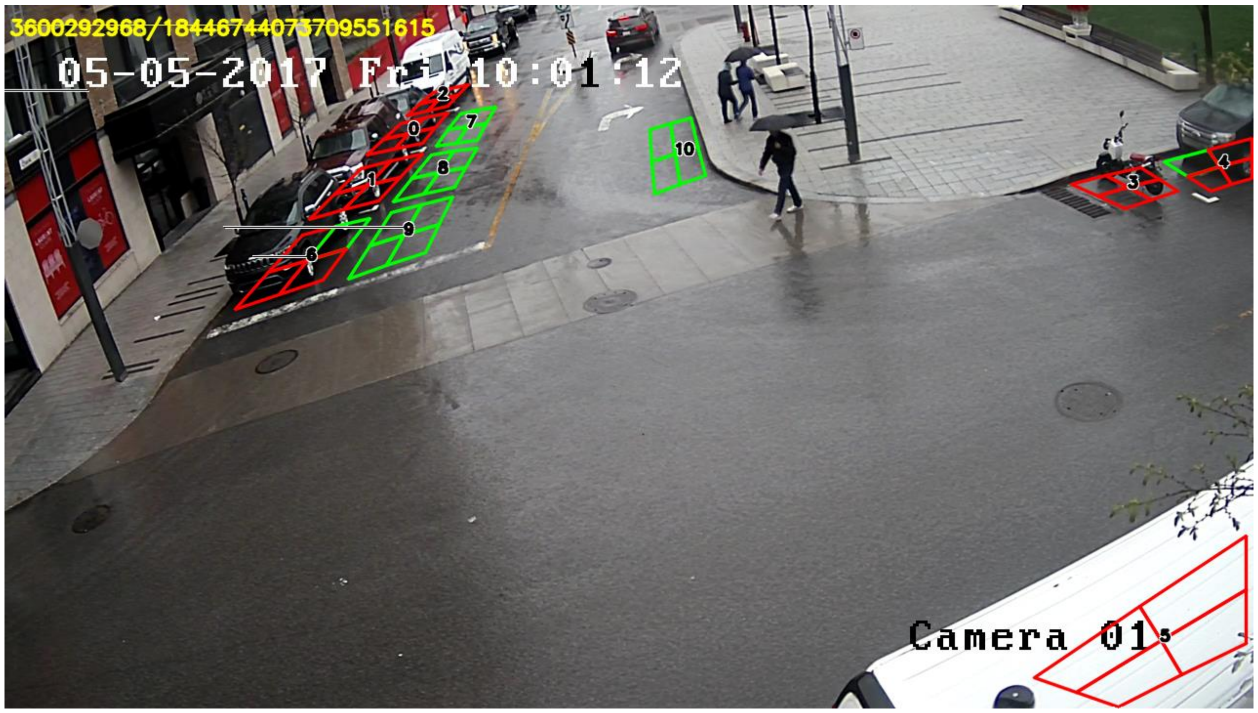


Figure 24: Performance of both algorithms combined

Each algorithm (Laplacian and luminance) applied individually result in a less than optimal detection. The edge-based algorithm does not provide reliable detection in highlights and shadows due to the lack of visibility of edges, and the luminance-based one is simply not systematic enough in detecting a vehicle (not every occupied parking spot will be overly bright or dark!) However, combining both algorithms results in a much more accurate detection, where the edge-based algorithm does most of the work, and the luminance-based one helps in cases where the edge algorithm performs poorly. The combination is done using the logical AND function: both algorithms must signal that a parking spot is free (1) for it to be considered free; if either algorithm marks it as occupied (0), it is automatically considered occupied. For a comparison of the performance of each algorithm and both algorithms combined, see Figure 21 and Figure 23 above, as well as Figure 24.

Notice the poor performance of edge-based parking detection in zones 4 and 6 (excessive shadows) and zone 5 (excessive highlights) (Figure 21), how the luminance-based parking detection algorithm detects those cases in particular (Figure 23), and how the combination of both algorithms produces the most accurate parking detection (Figure 24).

3.3.5 Defunct features

3.3.5.1 Manual contrast and brightness adjustment

In the early stages of developing the parking detection software, there was a necessity of modifying the contrast and brightness of the image to extract more detail from the highlights and shadows of the image. Before histogram equalization was discovered and explored, there was a method of adjusting the image contrast and brightness via user-defined values. Through research, it was found that the contrast is analogous to linear gain (factor by which each pixel's value is multiplied), and the brightness is a bias value (constant value by which each pixel's value is increased after contrast is applied) [18]. In short, if the image is represented by a function $f(i,j)$ where i and j are the pixel coordinates, then adjusting contrast (α) and brightness (β) generates a new image $g(i,j)$ computed as follows: $g(i,j) = \alpha \cdot f(i,j) + \beta$ [18]. However, this method was very difficult to fine-tune, due to the fact that two parameters must be adjusted simultaneously. Furthermore, it was not universal enough: the same parameters did not work for different times of day, since the base contrast and brightness of the image change based on ambient light (typically both are lower by night). An algorithm that varies the contrast and brightness parameters during program execution would then have to be implemented. This would be extremely unwieldy, so research was done on alternative methods of achieving the goal of extracting more detail from the image. This led to the finding of histogram equalization (CLAHE, see section 3.2.2), which is much easier to use (only requires adjusting the contrast limit) and is much more effective than anything that could be programmed from scratch given the limited time, resources, and knowledge of image processing available.

3.3.5.2 Canny edge detection

Before different edge detection thresholds for each parking spot were used (see section 3.3.1), the implementation of a reliable edge-based detection that would work with a single parameter was attempted. Research in this area led to the finding of the Canny edge detector developed by John F. Canny in 1986 and implemented as the function cv::Canny in OpenCV [19]. This algorithm has several steps, which allows it to improve on rudimentary edge detection techniques (such as the Laplacian, see section 3.3.2) in three key areas: it achieves a low error rate (only detect actual edges), good localization (a small distance between detected and actual edges), and minimal response (only detecting each edge once) [19]. The steps are as follows [19]:

1. Filter out noise using a Gaussian blur filter, where each pixel's value is replaced by the average of the values of the surrounding pixels weighted using a Gaussian distribution in the x and y directions, a typical kernel is described by *Formula 2*:
$$\frac{1}{159}\cdot\begin{pmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{pmatrix} \tag{2}$$
2. Find the gradient of the image by applying kernels similar to those in *Formula 3* (in the x and y directions) to each pixel:
$$G_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \tag{3}$$
Then, the norm (L_1: $|G_x| + |G_y|$ or L_2: $\sqrt{G_x^2 + G_y^2}$) and direction are computed for each pixel. This is done by the Sobel operator (cv::Sobel function with kernel size 3 in OpenCV) [12], which is also used by the Laplacian function [17].
3. Remove pixels that are not considered to be part of an edge, leaving only thin lines.
4. Apply hysteresis with an upper and lower threshold: any gradient above the upper threshold is a valid edge, any gradient below the lower threshold is not an edge and is rejected, any gradient between the two thresholds is only considered a valid edge if it is adjacent to a gradient above the upper threshold.

The Canny edge detector is obviously more capable than the Laplacian method: it has two additional steps (3 and 4) to improve reliability of edge detection. Nevertheless, after attempts to implement it, it was not used and the original Laplacian was kept. Indeed, the results were not as good as expected. First, the algorithm integrates the Gaussian blur, and the blur parameters cannot be modified (the cv::Canny function only takes as parameters the upper and lower thresholds, the Sobel kernel size, and a parameter indicating whether the L_1 or L_2 norm should be used). Since it was found that adding Gaussian blur is counterproductive in this project due to the already blurry video stream (see section 3.2.1), the Canny algorithm was not optimal. Furthermore, it was difficult to set the two thresholds to produce a good result, despite the recommendation to keep the upper/lower ratio between 2 and 3 [19]: either too few edges were detected, or too many. It became obvious that a variable edge threshold would need to be set for each parking spot, and once this feature was implemented, the Laplacian function worked well enough. It was then decided the Canny algorithm was not necessary and the Laplacian remained.

3.3.5.3 Chrominance-based detection algorithm

After the weaknesses of the edge-based detection algorithm became known (it completely failed in very bright or dark areas), it was obvious that a second detection algorithm would be needed to complement it. Before the luminance-based algorithm was implemented (see section 3.3.3), the possibility of implementing an algorithm based on the YUV color space (also called YCbCr) was investigated. Instead of encoding a pixel value as three values from 0 to 255 representing the amount of red, green, and blue (RGB color space), it represents it as a combination of luminance (Y) and chrominance (Cb and Cr). To convert from RGB to YUV, the luminance is obtained using *Formula 1* (section 3.2.1), and Cr and Cb are obtained using *Formula 4* and *Formula 5*, respectively [11]:

$$Cr \leftarrow (R - Y) \cdot 0.713 + 128 \tag{4}$$

$$Cb \leftarrow (B - Y) \cdot 0.564 + 128 \tag{5}$$

From these equations, the luminance is the same as the pixel value for the grayscale image, while the two chrominance channels represent the excess of red and blue in the image. The plan was to set three sets of upper and lower thresholds (one for each channel) and a parking spot would be considered occupied when the value of either channel is not between the two thresholds (which would allow the detection of excessive/insufficient brightness, an excess of red, or an excess of blue). Unfortunately, this did not work out as expected. When attempting to determine the thresholds, it was found that the difference between an occupied and free parking spot was very difficult to detect reliably using the chrominance, since most vehicles are not excessively red, nor blue. Furthermore, any additional image processing techniques that might be needed would take significantly longer, since they would have to be applied on three channels instead of one. Finally, only the luminance-based detection was kept, since it was helpful for highlights and shadows, unlike the chrominance one.

Another color space that was considered is HSV (hue, saturation, value). It encodes each pixel as three values representing the hue (the color of the pixel), saturation (the intensity of the color), and value (the lightness of the pixel). However, it was not used since color-based algorithms were found to be inefficient.

3.4 Motion detection image processing

The image processing pipeline in motion detection is quite complex. Nevertheless, one can see that it consists of three distinct steps: difference frame generation, shadow mask generation, and dilation/thresholding. Each of these steps is described in the section below, and the overall algorithm is shown in the following image (**Figure 25**):

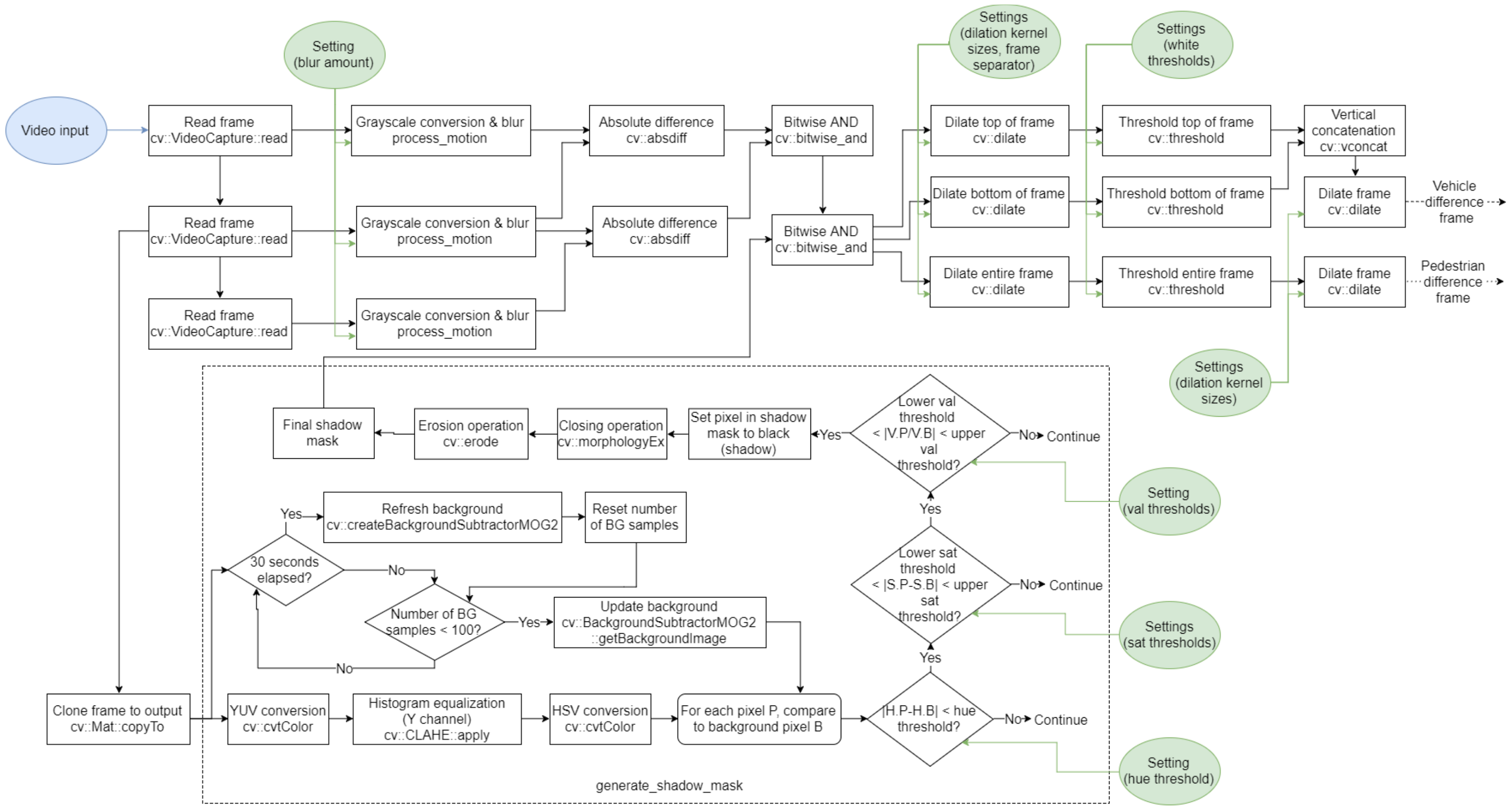

Video input
Read frame cv::VideoCapture::read
Read frame cv::VideoCapture::read
Read frame cv::VideoCapture::read
Setting (blur amount)
Grayscale conversion & blur process_motion
Grayscale conversion & blur process_motion
Grayscale conversion & blur process_motion
Absolute difference cv::absdiff
Absolute difference cv::absdiff
Bitwise AND cv::bitwise_and
Bitwise AND cv::bitwise_and
Settings (dilation kernel sizes, frame separator)
Dilate top of frame cv::dilate
Dilate bottom of frame cv::dilate
Dilate entire frame cv::dilate
Settings (white thresholds)
Threshold top of frame cv::threshold
Threshold bottom of frame cv::threshold
Threshold entire frame cv::threshold
Vertical concatenation cv::vconcat
Dilate frame cv::dilate
Dilate frame cv::dilate
Vehicle difference frame
Pedestrian difference frame
Settings (dilation kernel sizes)
Final shadow mask
Erosion operation cv::erode
Closing operation cv::morphologyEx
Set pixel in shadow mask to black (shadow)
Yes
Lower val threshold < |V.P/V.B| < upper val threshold?
No
Continue
Setting (val thresholds)
Yes
Lower sat threshold < |S.P-S.B| < upper sat threshold?
No
Continue
Settings (sat thresholds)
Yes
|H.P-H.B| < hue threshold?
No
Continue
Setting (hue threshold)
Yes
Refresh background cv::createBackgroundSubtractorMOG2
Reset number of BG samples
30 seconds elapsed?
No
Number of BG samples < 100?
Yes
Update background cv::BackgroundSubtractorMOG2 ::getBackgroundImage
No
Clone frame to output cv::Mat::copyTo
YUV conversion cv::cvtColor
Histogram equalization (Y channel) cv::CLAHE::apply
HSV conversion cv::cvtColor
For each pixel P, compare to background pixel B
generate_shadow_mask

Figure 25: Motion detection image processing pipeline

3.4.1 Difference frame

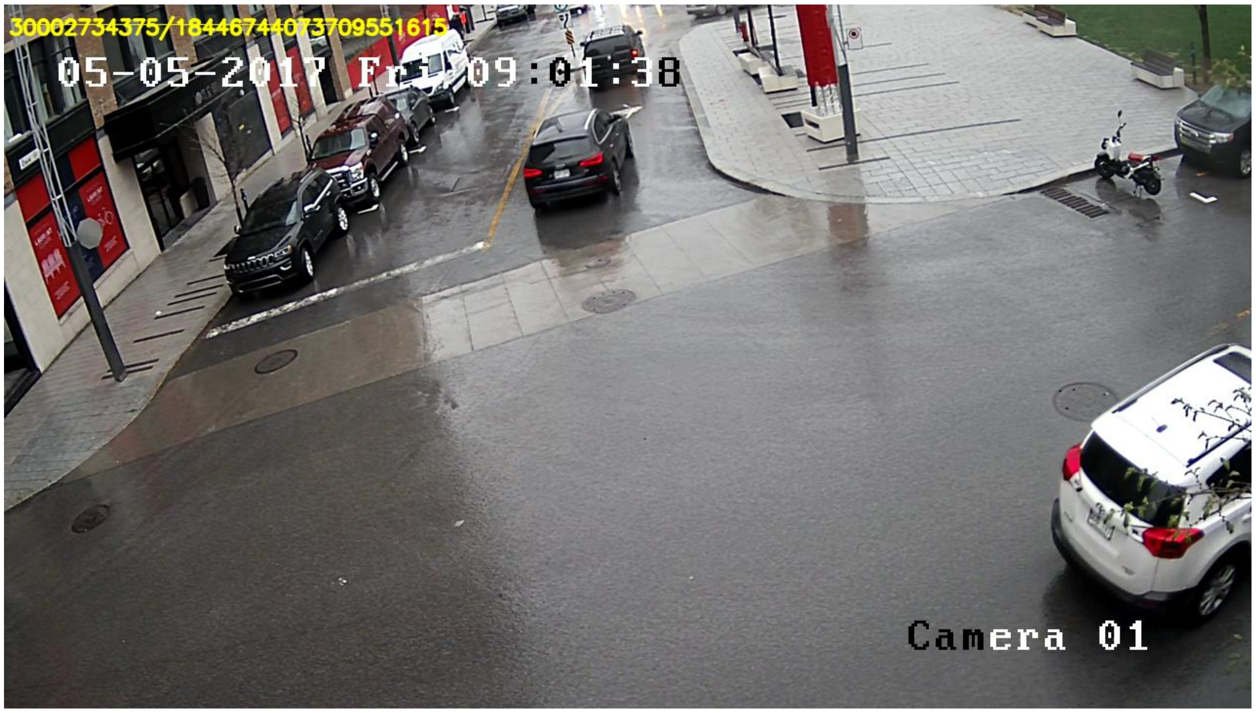


Figure 26: Original frame

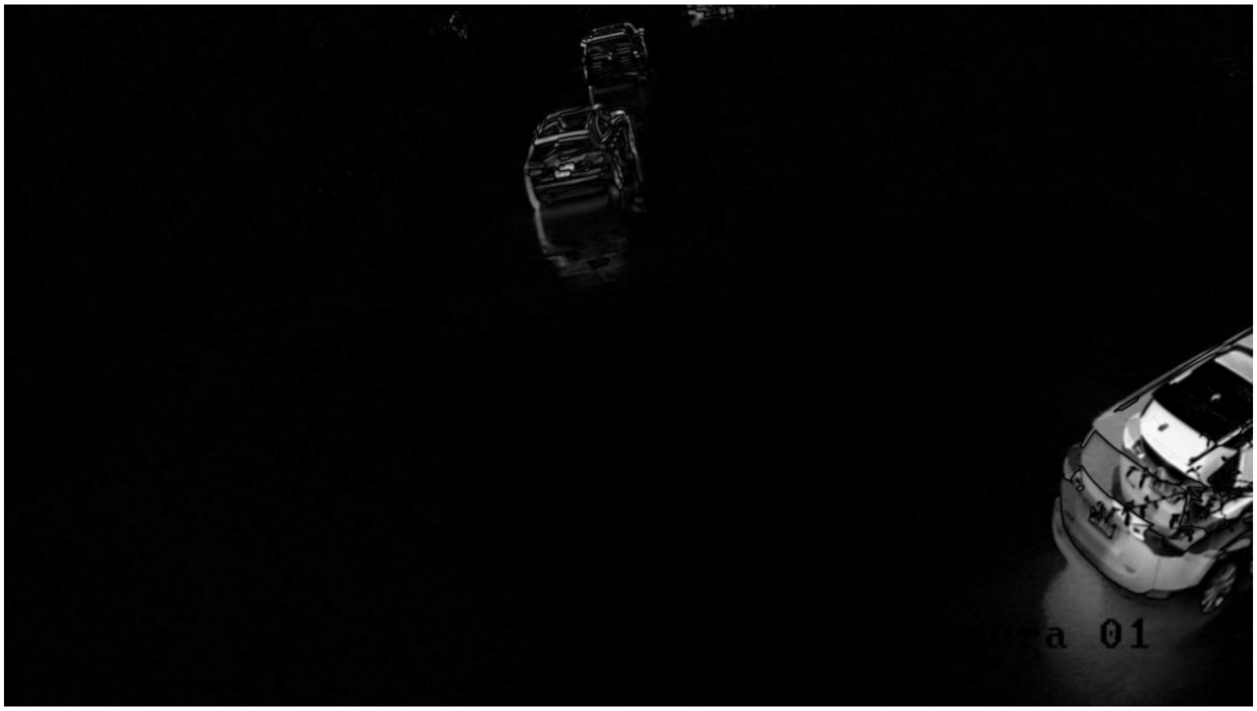


Figure 27: Frame after absolute difference operation

The first image processing step after acquiring a frame is conversion to grayscale, since color information is not necessary for motion detection [8][9][10][20]. Then, a given (user-defined) amount of Gaussian blur is applied to remove unnecessary details from the frame and thus reduce the amount of extraneous contours detected [9][20]. These steps are performed using function `process_motion`.

The next step is to determine the moving parts of the frame, i.e. to separate them from the static background [8][9][10]. Since the background is not constant in this case (due to the different possible

parked vehicles and ambient light level), this operation is performed by obtaining the consecutive frame, then taking the (absolute) difference between the current and previous frame [8]. In order to eliminate motion lasting only approximately one frame, which is considered noise, three consecutive frames can be used: I_{t-1}, I_t, I_{t+1} [10]. The difference frame can be computed using $| I_t – I_{t-1}| \wedge | I_{t+1} – I_t|$ (bitwise AND of the absolute difference between the current and previous frame, and absolute difference between the next and current frame) [10].
The following images show a reference frame (**Figure 26**), and the resulting difference frame computed using the previous and next frames (**Figure 27**).

3.4.2 Shadow mask

One of the main difficulties with the above approach to detecting motion (based on separating foreground from background) is the presence of shadows:

> "A more common approach for detecting people in a video sequence is to detect foreground pixels, for example via Gaussian mixture models [35, 40]. However, current techniques typically have one major disadvantage: shadows tend to be classified as part of the foreground. This happens because shadows share the same movement patterns and have a similar magnitude of intensity change as that of the foreground objects [30]." [21].

For the end goal of this project, counting, shadows pose a very significant problem because they cause distinct objects to be merged together in the contour detection algorithm (see section 3.5.3.1), since the shadow of an object can overlap another object. Therefore, a shadow detection algorithm was needed. After reading was done on different shadow detection algorithms based on various features of shadows (chromacity, physical/geometrical features, textures [21], and edges [22]), the chromacity-based algorithm described in [23] was chosen because it is the simplest to implement and produces great results according to [21].
The first step is to account for variations in ambient lighting which affect the appearance of shadows in the frame. To make the frames look more or less uniform regardless of time of day, histogram equalization (implemented using the OpenCV function `cv::Ptr<cv::CLAHE> clahe`), which was investigated in section 3.2.2, was used. Since only the luminance should be corrected, the frame is first converted RGB to the YUV color space, which separates brightness information (Y) and color information (U, V). Then, histogram equalization is applied to the Y channel, and the frame is converted back to RGB.
Since the chromacity algorithm requires comparing foreground pixels to the corresponding background pixels [21][23], a background image needed to be defined, which is very difficult in this application because the background changes quite often (see section 3.5.5.1). However, it turns out to be feasible because the background image does not need to be extremely accurate in this case, since it is not used to calculate motion, but simply as a reference for finding shadows. The background was obtained using the built-in OpenCV class `cv::BackgroundSubtractorMOG2`. This class implements a "Gaussian mixture-based background/foreground segmentation algorithm" [24], although only the background is relevant in this application. The background is built by deciding whether each pixel belongs to the foreground or background by a probabilistic decision, then the background model is updated using a training algorithm, which increases accuracy of future background/foreground segmentation [25]. The background subtractor must be applied to each frame to generate the foreground (and background) images [26], and the background image is then retrieved using the `getBackgroundImage()` method of the `BackgroundSubtractorMOG2` object [27]. The training parameters are specified by the `history` parameter, which specifies the number of past frames used for learning, and the `learningRate` parameter, which specifies the speed at which the background model is updated [24]. Learning the background at a slower rate, over a long period produces a more accurate image but is slower, while learning the background at a faster rate, over a shorter period produces a less accurate image quicker. A reasonably fast history (100 frames) was chosen and the learning rate was kept as default, which produced a usable background quickly, at the expense of accuracy: the algorithm learned the background

over a period of a few seconds only, so vehicles that were static over this period were integrated in the background. To solve this issue, the background was refreshed at a set interval (every 30 seconds) by recreating the `BackgroundSubtractorMOG2` object, to ensure the background is always representative of the current state of the scene.

Once a background has been defined, the foreground pixels are compared to background pixels to determine if they are part of a shadow or not [21][23]. To achieve this, the HSV color space is used instead of the traditional RGB: "we analyze pixels in the Hue-Saturation-Value (HSV) color space. The main reason is that the HSV color space explicitly separates chromaticity and luminosity and has proved easier than the RGB space to set a mathematical formulation for shadow detection" [23]. Indeed, the hue (encoded as an angle on a circular color wheel) represents which pure color a given pixel is closest to, disregarding tint, tone, and shade, the saturation represents where the pixels' color lies on the spectrum between white (saturation 0) and the corresponding pure color (saturation 1), and the value or lightness represents how dark a color is (with black corresponding to value 0) [28]. It is obvious that this color space is optimal for detecting shadows, since they should have very little impact on the hue, while decreasing value greatly: "Since the value (V) is a direct measure of intensity, pixels in the shadow should have a lower value than pixels in the background. Following the chromacity cues, a shadow cast on background does not change its hue (H)" [21].

The conditions for a pixel to be considered a shadow and included in a shadow mask (i.e. $SP(p) = 1$) are the following [23]:

$$SP(p)\begin{cases} 1\ if\ \alpha \le \frac{I(p).V}{B(p).V} \le \beta \wedge |I(p).S - B(p).S| \le \tau_s \wedge D_H(p) \le \tau_H \\ 0\ otherwise \end{cases} \quad (6)$$

$$D_H(p) = \min\{\, |I(p).H - B(p).H|,\ \ 360 - |I(p).H - B(p).H|\,\} \quad (7)$$

$\mathrm{I(p)}$ represents a pixel from the image (foreground) and $\mathrm{B(p)}$ represents the equivalent pixel from the background. The .H, .S, .V represent the hue, saturation, and value of the pixel. Essentially, *Formula 6* states that a pixel can be considered part of a shadow if the ratio between its value and the background value is between two thresholds α and β, the absolute difference between its saturation and the background saturation is below a threshold τ_s, and the absolute difference between its hue and the background hue (computed as an angular value using *Formula 7*) is below a threshold τ_H. Note that the definition of shadow mask was reversed: pixels start as white (binary 1) and if shadows are detected, they are turned black (binary 0). This is because the shadow mask will be applied to the difference frame via the bitwise AND function. A lower threshold for saturation was also introduced to achieve better results: the condition for a pixel to be considered a shadow in terms of its saturation is now $\tau_{sl} \le |I(p).S - B(p).S| \le \tau_{sh}$.

After the first version of the mask is generated, the closing morphological operation of the mask is applied on it. This operation is "useful in closing small holes inside the foreground objects, or small black points on the object" [29], and small groups of black pixels need to be eliminated from the mask, since they are likely not part of a shadow. Then, the erosion operation is applied to the object. Erosion consists in applying a kernel to the image and, for each pixel, replacing its value by the minimum value over the kernel [30]. This has the effect of eroding, or decreasing the width of white objects on a black background [30]. Alternatively, it dilates, or increases the width of black objects on a white background [30], which has the effect of connecting multiple parts of the shadow mask into one contiguous shadow and increases the width of shadows slightly to improve the reliability of the mask. The type of the kernels used for these two operations, `close_kernel` and `erode_kernel`, should also be noted: in this case, it is an ellipse. Other types that could be used are the rectangle and the cross. The shapes of different kernels (5x5) are shown below [29], which explains why the ellipse was chosen: it should provide the smoothest result, without ragged edges.

$$\begin{array}{ccccc} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{array} \qquad \begin{array}{ccccc} 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{array} \qquad \begin{array}{ccccc} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array}$$

Rectangular kernel Elliptical kernel Cross kernel

To speed up iterating over the numerous subpixels of each frame (1280 · 720 · 3 = 2764800, or nearly 3 million for a 720p image), multidimensional array vectoring by rows was used, where the base addresses of the rows are stored in a pointer [31]:

```
uchar *p = frame_hsv.ptr(i);
uchar *q= bg_hsv.ptr(i);
uchar *r = shadow_mask.ptr(i);
```

Then, the column addresses are incremented every time the pointer is dereferenced to access the corresponding subpixel [31]:

```
int hue_t = cv::abs((*p++) - (*q++));
int sat = cv::abs((*p++) - (*q++));
int val_f = *p++;
int val_b = *q++;
```

This increases efficiency significantly over using a syntax such as `frame_hsv[i][j]` because it saves a multiplication and addition at every access.

The following images show a reference frame (**Figure 28**), the initial shadow mask (**Figure 29**), the shadow mask corrected by morphological operation (Figure 30), the initial difference frame (**Figure 31**), and the difference frame after application of the shadow mask (**Figure 32**).

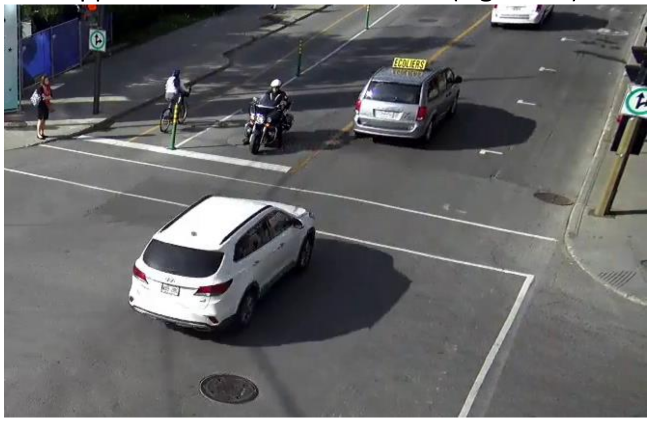


Figure 28: Reference frame used to demonstrate shadow detection

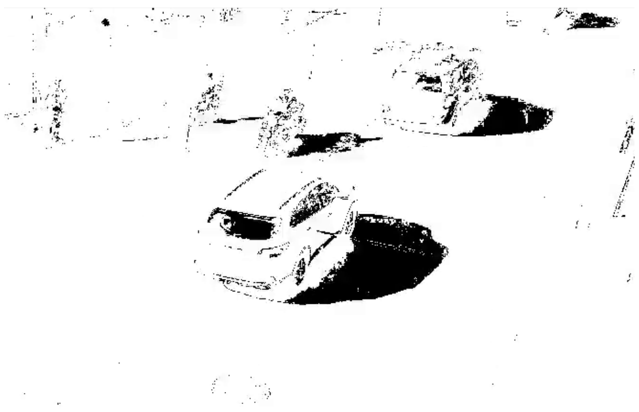


Figure 29 Initial shadow mask (before morphological operations)

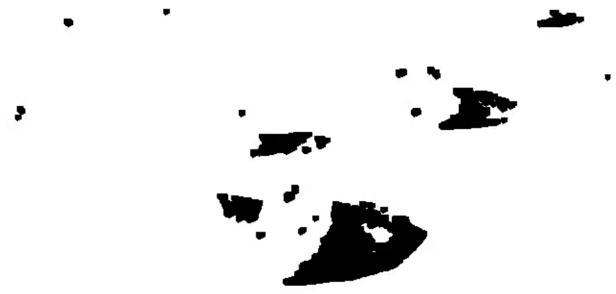


Figure 30 Final shadow mask (after morphological operations)

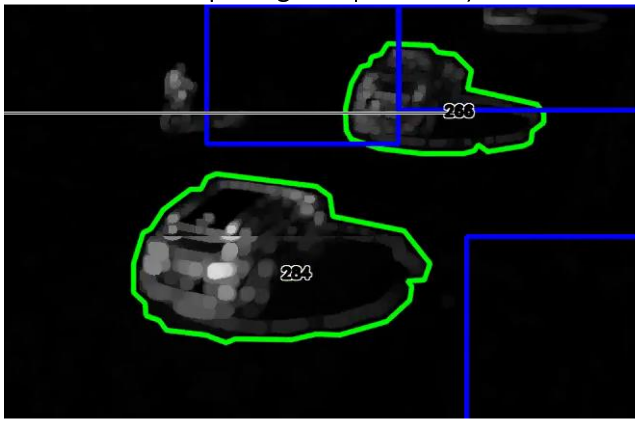


Figure 31: Initial difference frame (before application of shadow mask)

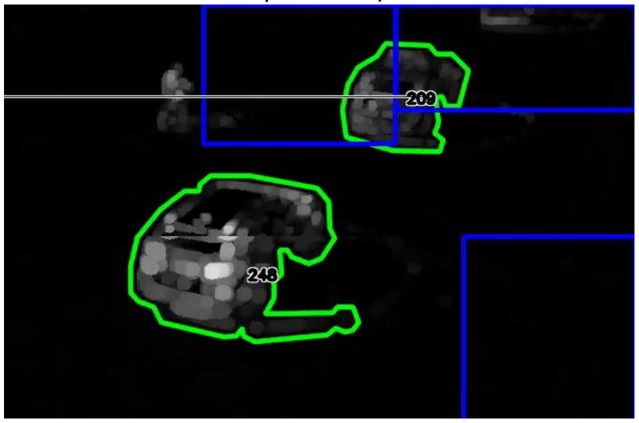


Figure 32: Final difference frame (after application of shadow mask)

From these images, one can see that the initial shadow mask adequately represents the shadows of moving objects, but also has a lot of noise (individual pixels or small groups of pixels that are detected as a shadow). After the closing and dilation operations, the final shadow mask has been smoothed significantly: the noise is mostly gone, but the contours of shadows are less precise. However, this is not an issue: looking at the initial and final difference frames, the application of the shadow mask was effective enough to remove the entirety of the shadow from one vehicle and most of it from the other. That is because the mask does not need to cover the shadow entirely for proper removal: as long as the shadow is no longer contiguous to the object, it will not be detected by the object detection algorithm (see section 3.5.3.1).

3.4.3 Dilation and thresholding

The difference frame alone is still not suitable because there are too many small details; in particular, single large moving objects are represented by a multitude of small shapes, which is unsuitable. Therefore, two more steps are performed next: dilation and thresholding [8][9][20]. Dilation is the process of applying a kernel to the image and, for each pixel, replacing its value by the maximum value over the kernel [30]. This has the effect of dilating, or increasing the width of white objects on a black background. Similarly to the morphological closing and erosion used for the shadow mask (see section 3.4.2), the kernel type is the ellipse, with a user-defined size. Although dilation causes large objects to appear as a single entity, it is still difficult for an algorithm to find the exact edge of such objects, given that they are somewhat blurry. Thus, the thresholding operation is applied, which converts all pixel values above a user-defined threshold to 100% white (value 255) and leaves the others black [32]. After, another dilation operation is performed to close most remaining gaps. The result after thresholding is that moving objects have a clearly defined shape and contour.

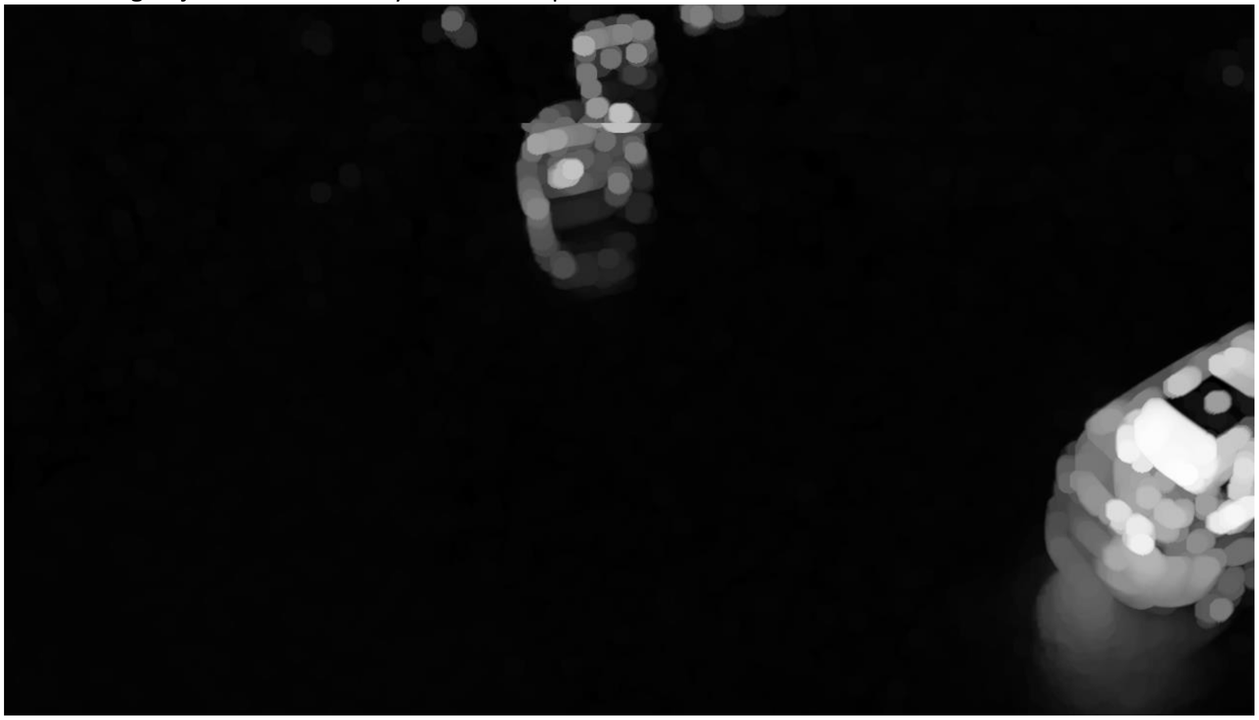


Figure 33: Difference frame after dilation operation

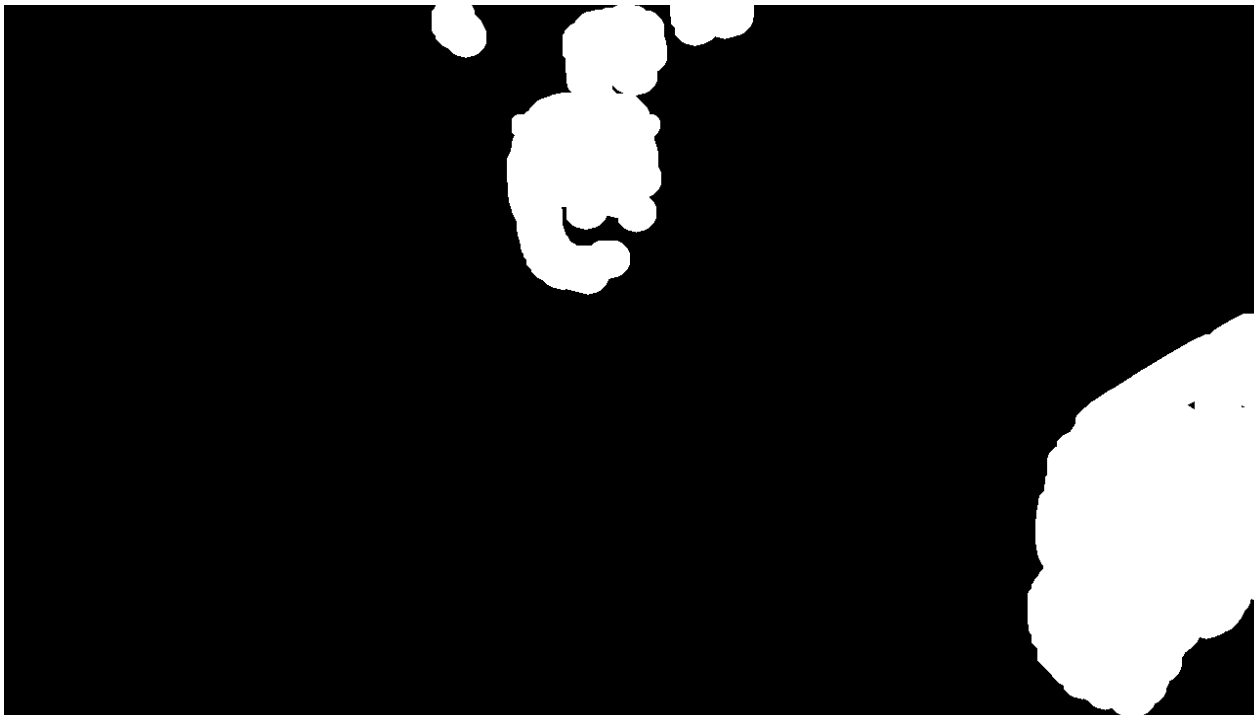


Figure 34: Difference frame after dilation and thresholding operations

Separate dilation and thresholding operations are performed for "large objects" (vehicles) and "small objects" (pedestrians). That is because the parameters that are optimal for accurate vehicle detection do not work well for pedestrians: they tend to be recognized as multiple distinct objects. Therefore, a larger dilation kernel size (`element_pede`) and a lower threshold for casting to white (`white_thresh_pede`)

are required for detecting pedestrians accurately. Furthermore, for the vehicle detection, dilation and thresholding are performed separately on the top and bottom part of the frames, then they are concatenated together. That is because objects are much smaller and details are finer at the top of the frame, so different kernels and threshold values are used for the dilation (`element_top` is smaller than `element_bot`, and `white_thresh_top` is higher than `white_thresh_bot`).
The following images show the results of dilation (**Figure 33**) and thresholding (**Figure 34**) when applied to the difference frame shown in Figure 27 (section 3.4.1).

3.5 Motion detection features and algorithms

3.5.1 The ObjectContour object

In order to perform the tracking and counting of vehicles and pedestrians in the frame, there must be a way to relate the contour of a given object from one frame that object's contour from the previous frame. Furthermore, it would be incredibly useful for each object to carry information about its past state, such as the area where it entered the frame, the number of frames it has existed for, etc. To regroup all this information, an `ObjectContour` class was defined. Its objects represent pedestrians or vehicles detected in the frame, and carry all aforementioned information in their attributes. The class is defined as follows:
The parameters and their associated methods are the following:

- `vector<cv::Point> contour`: a vector of points representing an object's contour (extracted from the difference frame).
  - Getter method: `vector<cv::Point> getContour(void)`
- `cv::Rect bounding_rect`: the bounding rectangle of the ObjectContour's contour
  - Getter method: `cv::Rect getBoundingRect(void)`
- `double area`: a numerical value for the area of the ObjectContour's contour
  - Getter method: `double getArea(void)`
- `cv::Poin2f center`: the point representing the center of the ObjectContour's contour
  - Getter method: `cv::Point2f getCenter(void)`
- `cv::Poin2f center_orig`: the point representing the center of the first contour assigned to a particular ObjectContour, used to calculate the distance an ObjectContour traveled across the frame
  - Getter method: `cv::Point2f getCenterOrig(void)`
  - Setter method: `void setCenterOrig(cv::Point2f n)`
- `int id`: a number used to identify each ObjectContour uniquely, used to link two ObjectContour objects (by assigning them the same ID)
  - Getter method: `int getId(void)`
  - Setter method: `void setCenterOrig(cv::Point2f n)`
- `int origin`: a number used to identify where a given ObjectContour entered the frame (default value: -1, left side: 0, top side: 1, right side: 2, bottom side: 3)
  - Getter method: `int getOrigin(void)`
  - Setter method: `void setOrigin(int n)`
- `int end`: a number used to identify where a given ObjectContour exited the frame (default value: -1, left side: 0, top side: 1, right side: 2, bottom side: 3)
  - Getter method: `int getEnd(void)`
  - Setter method: `void setEnd(int n)`
- `int lifetime`: a number used to determine how long (how many frames) an ObjectContour has existed, used for filtering out false detections (objects appearing for a frame and disappearing immediately after)
  - Getter method: `int getLifeTime(void)`
  - Setter method: `void setLifeTime(int n)`

- `int exit_status`: a value used to determine whether an ObjectContour has in fact disappeared from the frame (default value: -1, disappeared for 1 frame (candidate for disappearing): 0, disappeared for 2 consecutive frames (definitely disappeared): 1)
    - Getter method: `int getExitStatus(void)`
    - Setter method: `void setExitStatus(int n)`

The constructor for ObjectContour objects requires a vector of points representing an object's contour (`contour_poly`) and an integer n, which represents an ID value. To define the ObjectContour, the contour is set to `contour_poly`, and the bounding rectangle, area, and center are calculated from that contour. Then, the id is set to the provided value and the origin, end, lifetime, and exit status are set to the default (initial) values (origin, end, and exit status to -1, lifetime to 0).

3.5.2 Perspective correction

One of the main challenges when working with object detection was to properly define a minimum (and perhaps maximum) size of object that would be considered as a vehicle or pedestrian. Constant thresholds do not work because of the perspective of the frame: the same object will be much bigger near the camera than further away: if only thresholds were used, there would be either too many invalid (small) objects near the camera that are detected, or too many valid (large) objects far from the camera that are not detected. A way to deal with the perspective is by using the following relationships:

$$(origin.x, origin.y) = (h_{offset} \cdot frame_{width}, v_{offset} \cdot frame_{height}) \tag{8}$$

$$(center.x', center.y') = (h_{multi} \cdot center.x, v_{multi} \cdot center.y) \tag{9}$$

$$(origin.x', origin.y') = (h_{multi} \cdot origin.x, v_{multi} \cdot origin.y) \tag{10}$$

$$Dist.threshold = gain \cdot (offset - \|(center.x', center.y') - (origin.x', origin.y')\|) \tag{11}$$

$$Area\ threshold = gain \cdot (offset - \|(center.x', center.y') - (origin.x', origin.y')\|)^2 \tag{12}$$

The definition of the variables is the following: *gain* is a constant scaling parameter, *offset* is the threshold value (before gain) associated with an object that is as close as possible to the camera, *(center.x, center.y)* are the coordinates of the center of the object, and *(origin.x, origin.y)* are the coordinates of the origin of the perspective, or the point where objects will appear the largest in the frame. *(origin.x, origin.y)* are derived from the frame width multiplied by a parameter h_{offset} and the frame height multiplied by a parameter v_{offset} (*Formula 8*). *(center.x', center.y')* and *(origin.x', origin.y')* , computed using *Formula 9* and *Formula 10* respectively, are the transformed equivalents of the above: the x-coordinate is multiplied by h_{multi}, and the y-coordinate is multiplied by v_{multi} to account for the fact that a horizontal change of a given distance will lead to a much smaller perspective change than the equivalent distance change in the vertical direction. Finally, the one-dimensional (distance) and two-dimensional (area) thresholds can be computed using *Formula 11* and *Formula 12*.

Parameters *gain*, *offset*, h_{multi}, v_{multi}, h_{offset}, v_{offset} are user-defined and should be chosen with care to avoid errors. Notably, the maximum scaled distance between any point in the frame and the origin $\|(center.x', center.y') - (origin.x', origin.y')\|)$ should be smaller than *offset* to avoid the possibility of a negative distance threshold, which would be indicated by missing rectangles in the perspective verification (section 3.1.3).

The effects of perspective correction can be seen by enabling perspective verification, which shows the relative size of an object depending on its position in the frame (displayed using rectangles). This allows the user to directly view the effect of their changes of the various perspective parameters and eliminate the guessing factor in obtaining correct perspective correction. Figure 8 in section 3.1.3 shows an example of perspective verification.

3.5.3 Object detection, tracking, and counting algorithms

3.5.3.1 Object detection

After the absolute difference frame is generated and dilation/thresholding operations are performed, objects' contours can be detected. This consists of two steps: a first function `locate_contours` finds all contours in the frame, and a second function `remove_contours` removes uninteresting contours that do not correspond to an object being tracked. **Figure 35** shows the object detection algorithm:

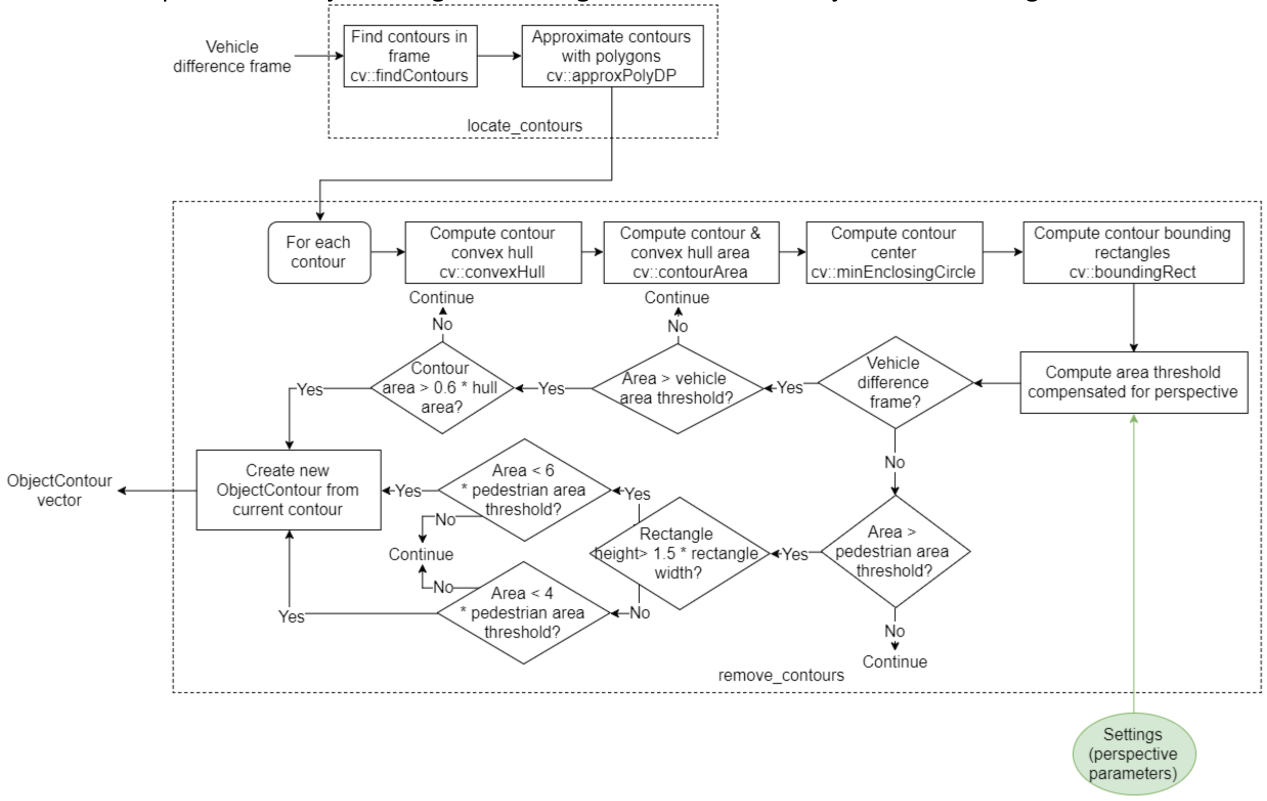

Vehicle difference frame
Find contours in frame cv::findContours
Approximate contours with polygons cv::approxPolyDP
locate_contours
For each contour
Compute contour convex hull cv::convexHull
Compute contour & convex hull area cv::contourArea
Compute contour center cv::minEnclosingCircle
Compute contour bounding rectangles cv::boundingRect
Compute area threshold compensated for perspective
Vehicle difference frame?
Area > vehicle area threshold?
Contour area > 0.6 * hull area?
Continue
No
Yes
Create new ObjectContour from current contour
ObjectContour vector
Area < 6 * pedestrian area threshold?
Rectangle height> 1.5 * rectangle width?
Area > pedestrian area threshold?
Area < 4 * pedestrian area threshold?
remove_contours
Settings (perspective parameters)

Figure 35: Object detection algorithm

Locating all the contours is achieved through the built-in function `cv::findContours()` [8][9][20]. This function takes as arguments an image from which the contours are to be extracted (in this case, the result after thresholding) and a vector of point vectors to store the detected contours (each element of this vector represents a list of points that forms a contour) [33]. It also has optional arguments: a contour hierarchy to store the level of nesting of each contour (to determine which contours are inside other contours), a contour retrieval mode, a contour approximation method, and an offset (here, unnecessary, thus set to 0) [33].

The contour retrieval mode determines which contours are retrieved and how the contour hierarchy is built (for example, `CV_RETR_LIST` retrieves all contours and does not establish a hierarchy, `CV_RETR_TREE` retrieves all contours and organizes the hierarchy based on the level of nesting, and `CV_RETR_EXTERNAL` only retrieves external contours and ignores all contours nested within [33]). For this application, `CV_RETR_EXTERNAL` was the optimal choice because only one contour per object must be detected, so the external contour is all that is needed.

The contour approximation method determines which contour points are stored (for example, `CV_CHAIN_APPROX_NONE` stores every pixel that forms the contour, `CV_CHAIN_APPROX_SIMPLE` only stores end points of line segments, and `CV_CHAIN_APPROX_TC89_L1` is the Teh-Chin algorithm developed by C.H. Teh and R.T. Chin in 1989 [33]). The Teh-Chin algorithm was chosen for this application because it is more efficient than storing all points or only the vertices of straight lines, so it would lead to reduced memory usage for storing the contours. For more details on this algorithm, see [34].

After contours have been found, they are approximated to polygons using the built-in function `cv::approxPolyDP()` [20]. This function takes as arguments two vectors of points (for the original and approximated contours, a parameter corresponding to the maximum distance between a point on the original and approximated contour (set to 3) and a Boolean parameter specifying if the contour must be a closed shape (set to true) [33]. This simplifies the contours and thus allows even fewer points to be stored, speeding up any further calculations. This function uses the Douglas-Peucker algorithm, which is described in more detail in [35].

After the polygonal contours have been defined, it must be determined whether each contour corresponds to a pedestrian, a vehicle, or an irrelevant object (discarded). For vehicles, the selection is done based on the area of the contour: objects larger than a certain threshold (`gain`) are considered as vehicles. However, this poses problems when pedestrians cast large shadows: their contour area can be as big as a vehicle's. However, a key differentiator between the contours of vehicles and pedestrians casting large shadows is the convexity. Vehicles tend to be almost convex, with few small concave corners, while pedestrians and shadows have an "L" shape which is very concave. It thus follows that a vehicle's contour should be almost the same as its convex hull, which is the smallest convex contour that contains the original contour [33], while the pedestrian's contour will be very different from its hull. The convex hulls of each contour are computed using the built-in function cv::convexHull(), which takes as argument two vectors of points (for the original contour and its convex hull) and computes the convex hull. Contours' areas are also compared to those of their convex hulls and only if a contour has 60% of the area of its convex hull can it be considered to belong to a vehicle.

To determine which objects are pedestrians, an algorithm taking into account contour area and ratio between width and height is used. Objects between two thresholds (`gain_pede` and `4·gain_pede`) with a height of at least 0.8 times their width are considered as pedestrians, and the upper threshold is extended to `6·gain_pede` if the contour is 1.5 times taller than wider, since this shape is very typical of a human being. The two gain parameters are used-defined and adjustable.

If a contour is detected as a vehicle or pedestrian, an `ObjectContour` object (see section 3.5.1) is defined) based on this contour, and is assigned a unique ID via `global_id++.` This `ObjectContour` is then added to a vector (`small_contours_poly` for pedestrians, or `contours_poly` for vehicles).

3.5.3.2 Object tracking

Object tracking requires contour information from both the current frame and previous ones. Therefore, contours from the last three frames are stored in a vector of vector of `ObjectContour` objects (each inner vector corresponds to contours from a given frame). The previous frame contours are defined and updated in the manner shown below. Note that contours from up to three frames behind are used to track both vehicles and pedestrians (more than this leads to errors where two distinct objects may be considered the same, and fewer than this is not enough information).

Before tracking vehicles across multiple frames, nearby contours from each frame must be combined into a single contour. Indeed, even though a significant amount of dilation is performed (see section 3.4.3), large vehicles such as buses or trucks can still be detected as two, three, or more individual objects, which is unsuitable. This problem does not occur with pedestrians, so this step is only performed for vehicle detection. **Figure 36** below shows the contour combination algorithm steps, and they are then explained.

To combine contours, applying more dilation is not suitable because the two or more contours corresponding to a large object may be very far apart (several dozen pixels), so applying such a large amount of dilation will absolutely annihilate the precision of contour detection. Therefore, a more intelligent algorithm for performing contour combination was devised. Instead of using dilation to connect contours, it computes their convex hull using the built-in function `cv::convexHull()`.This is preferable to dilation because although two contours can be somewhat far from each other, one may "fit" into a gap of the other, so their hulls have an obvious overlap. In some cases, however, this is not sufficient, so this algorithm also uses contours from the previous frame to compute overlap. If two contours' hulls from the current frame overlap each other, or both overlap a contour hull from the previous frame, the contours in question are merged into a single one.

The algorithm iterates over all combinations of two contours, which are then checked for overlap if both contours are located in the bottom part of the frame. That is because contour combination should absolutely not be performed at the top part of the frame, since it would create more problems than it would solve (cars following each other in close proximity being combined). To check two contours for overlap, a mask is defined for each contour's convex hull. This mask is a black (binary 0) image upon which the convex hull is drawn and filled with white (binary 1). Then, a bitwise AND operation is performed on the two matrices and the resulting matrix, `overlap`, can be used to determine whether there is any overlap or not. If the mean value of this matrix is bigger than 0, it contains some white pixels: overlap is found.

If there is no overlap between two contours from the current frame, contours from the previous frame can be used to "fill the gap" between the two contours. The two contours in question are checked for overlap with every single contour from the previous frame. Overlap is determined in the same way as the previous case: by defining masks for each convex hull, using bitwise AND, and computing the mean of the resulting matrix. The `overlapping` flag is set and the iteration stops if both contours from the current frame have some overlap with any contour from the previous frame.

If the `overlapping` flag has been set by any previous operation, a new contour that contains points from both overlapping contours is defined. To avoid contour self-intersection (which leads to wrong results when calculating area [33]), only points from one contour that are not in the bounding rectangle of the other (and vice-versa) are added to the new contour. The new contour also gains the ID of one of the old contours. Then, the two old contours are removed from the `ObjectContour` vector and the new combined contour is added.

After contours have been combined in the case of vehicles, contour matching can occur between contours from the previous frame and new ones that appeared in the current frame. This is performed using a `match_contours` function, whose algorithm is shown in **Figure 37** below and detailed in the following paragraphs.

Specifically, for each contour in the current frame, the matching algorithm iterates over contours from the previous frame, searching for the closest contour that is below a certain minimum distance threshold. Only one old contour can be matched with every new contour. Priority is given to old contours that have been matched previously (lifetime parameter greater than 1) and have not been marked as exited from the frame (exit status parameter equal to -1). Priority is granted by allowing an old contour and a new contour to be matched even if another old contour with a shorter distance to the given new contour has been matched with it. When a matching contour is found, the index values are set to the contour's position in the old `ObjectContour` vector. If a contour satisfying these conditions is found (index values are a valid array position), the contours are linked. The values of the ID, the origin, and the exit point are copied from the old contour to the new, and the new contour is given the life time value of the old one incremented by one frame. Furthermore, this contour's ID is added to a `contour_id` vector storing the IDs of all contours that have been matched with another.

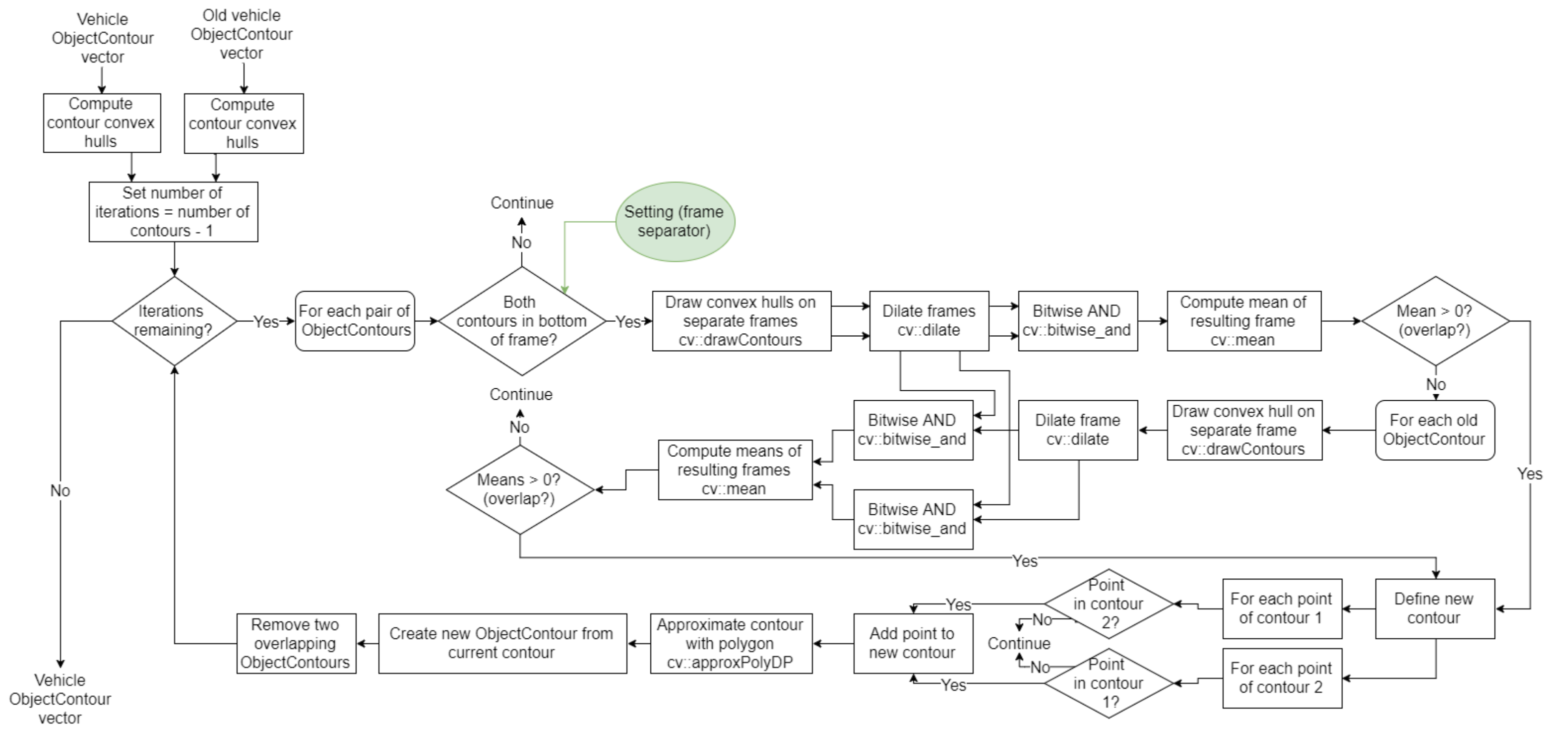

Vehicle ObjectContour vector
Old vehicle ObjectContour vector
Compute contour convex hulls
Compute contour convex hulls
Set number of iterations = number of contours - 1
Iterations remaining?
Yes
No
Vehicle ObjectContour vector
For each pair of ObjectContours
Both contours in bottom of frame?
No
Continue
Setting (frame separator)
Yes
Draw convex hulls on separate frames cv::drawContours
Dilate frames cv::dilate
Bitwise AND cv::bitwise_and
Compute mean of resulting frame cv::mean
Mean > 0? (overlap?)
No
Yes
For each old ObjectContour
Draw convex hull on separate frame cv::drawContours
Dilate frame cv::dilate
Bitwise AND cv::bitwise_and
Bitwise AND cv::bitwise_and
Compute means of resulting frames cv::mean
Means > 0? (overlap?)
No
Continue
Yes
Define new contour
For each point of contour 1
For each point of contour 2
Point in contour 2?
Point in contour 1?
Yes
No
Continue
Add point to new contour
Approximate contour with polygon cv::approxPolyDP
Create new ObjectContour from current contour
Remove two overlapping ObjectContours

Figure 36: Contour combination algorithm

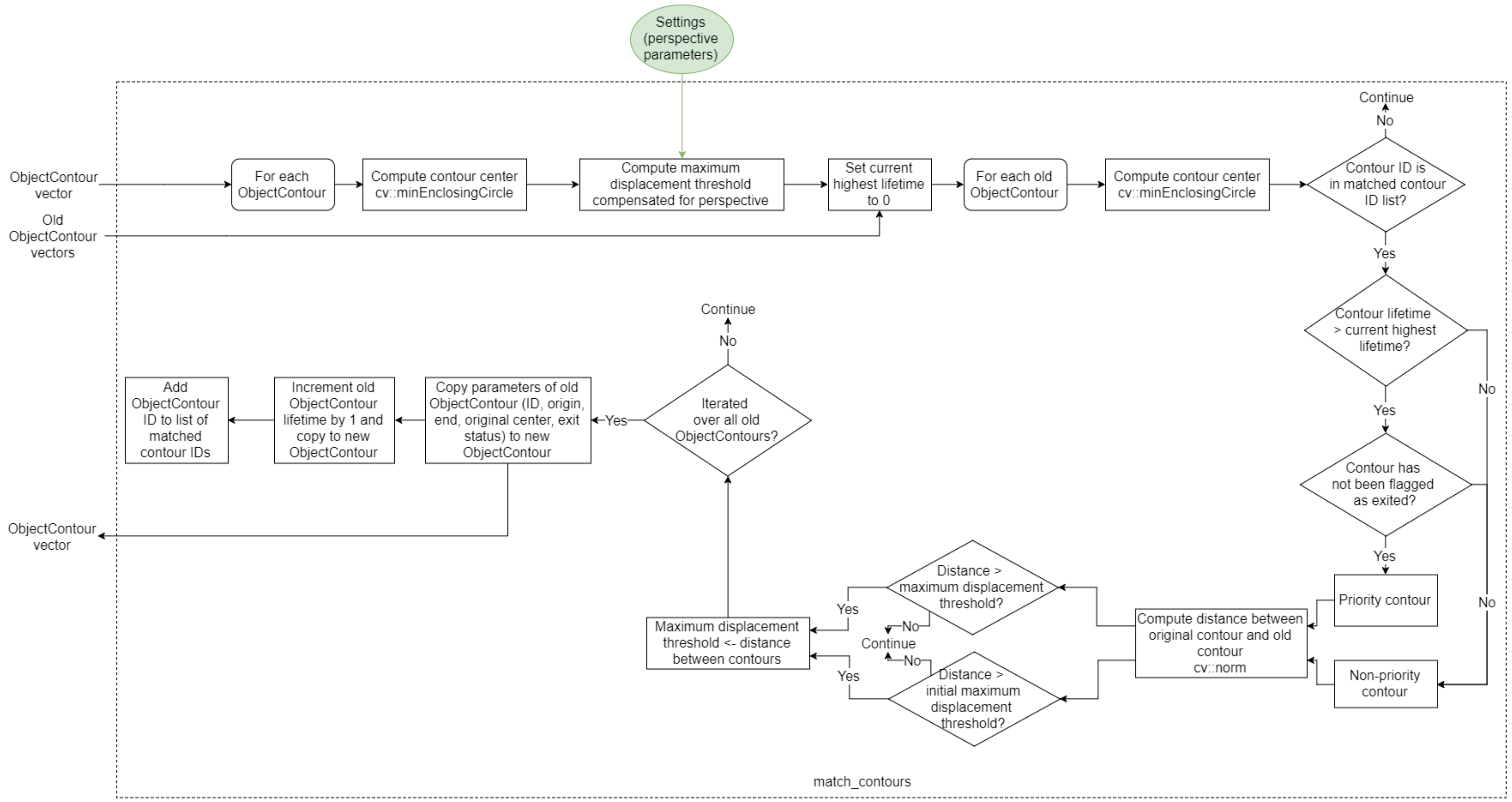

Settings (perspective parameters)
ObjectContour vector
Old ObjectContour vectors
For each ObjectContour
Compute contour center cv::minEnclosingCircle
Compute maximum displacement threshold compensated for perspective
Set current highest lifetime to 0
For each old ObjectContour
Compute contour center cv::minEnclosingCircle
Contour ID is in matched contour ID list?
No
Continue
Yes
Contour lifetime > current highest lifetime?
Yes
No
Contour has not been flagged as exited?
Yes
No
Priority contour
Non-priority contour
Compute distance between original contour and old contour cv::norm
Distance > maximum displacement threshold?
Distance > initial maximum displacement threshold?
Yes
No
Continue
No
Yes
Maximum displacement threshold <- distance between contours
Iterated over all old ObjectContours?
No
Continue
Yes
Copy parameters of old ObjectContour (ID, origin, end, original center, exit status) to new ObjectContour
Increment old ObjectContour lifetime by 1 and copy to new ObjectContour
Add ObjectContour ID to list of matched contour IDs
ObjectContour vector
match_contours

Figure 37: Contour matching algorithm

3.5.3.3 Object counting

So far, the motion detection algorithms (object detection and tracking) have been fairly similar for both vehicles and pedestrians. However, for counting, the algorithm used for the two classes of objects differs significantly. This is due to the different behavior of vehicles and pedestrians related to their motion in the frame. Vehicles must enter from a designated area (road) and most of them will exit through another such designated area, with a very small minority remaining in the frame (parking). Pedestrians, on the contrary, tend to appear and disappear anywhere in the frame because they either stop and restart moving or enter and exit buildings frequently.

The first step for both algorithms is the same: to check whether there are any contours that just appeared or just disappeared, which is done by pairwise comparisons. To determine if a contour just appeared, each new contour's ID value are compared to all old contours' IDs, and if no match is found, `new_contours` is incremented. Similarly, to determine if a contour just disappeared, each old contour is compared to all new contours, and `del_contours` is incremented if there is no match. The `calculate_diff` function executes these steps; its algorithm is shown in the image below (**Figure 38**).

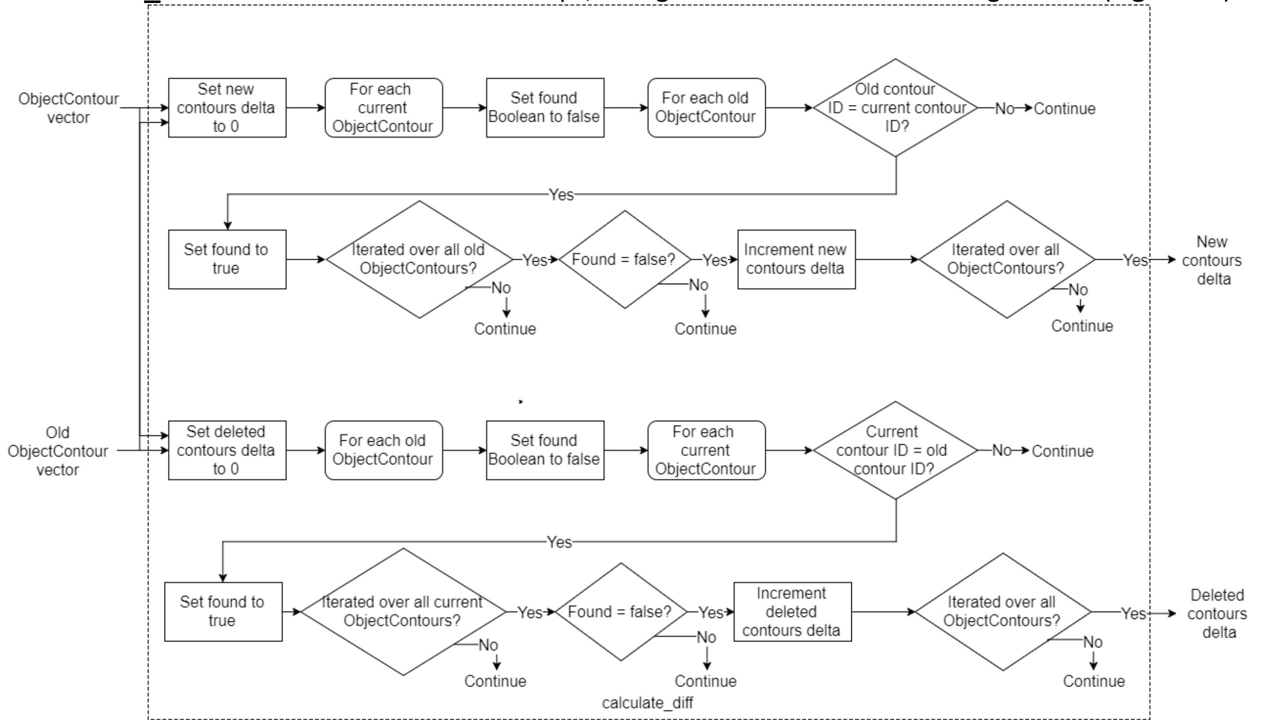

ObjectContour vector
Set new contours delta to 0
For each current ObjectContour
Set found Boolean to false
For each old ObjectContour
Old contour ID = current contour ID?
No
Continue
Yes
Set found to true
Iterated over all old ObjectContours?
Yes
No
Continue
Found = false?
Yes
No
Continue
Increment new contours delta
Iterated over all ObjectContours?
Yes
No
Continue
New contours delta
Old ObjectContour vector
Set deleted contours delta to 0
For each old ObjectContour
Set found Boolean to false
For each current ObjectContour
Current contour ID = old contour ID?
No
Continue
Yes
Set found to true
Iterated over all current ObjectContours?
Yes
No
Continue
Found = false?
Yes
No
Continue
Increment deleted contours delta
Iterated over all ObjectContours?
Yes
No
Continue
Deleted contours delta
calculate_diff

Figure 38: Contour counting (identification) algorithm: counting new/deleted contours

From this point on, the two algorithms (vehicle and pedestrian detection) are different and the two approaches are outlined below.

3.5.3.3.1 Vehicle counting

Once contours have been matched to ones in the previous frame, the process of counting can occur. Specifically, this portion of the program is concerned with detecting when and where vehicles enter or exit the frame (through designated entrance and exit zones at the left, top, right, or bottom areas of the frame). It also keeps track of the path of each individual vehicle whenever possible. Unfortunately, the program can sometimes fail to keep track of a particular vehicle if it stops in the middle of the frame or is obscured by another vehicle. However, this does not affect the count of vehicles passing through each entrance or exit, so it is not a major issue.

If there are new contours, their point of origin must be determined, which is done using the contour entering/exiting identification algorithm in **Figure 39** below, which is explained further down.

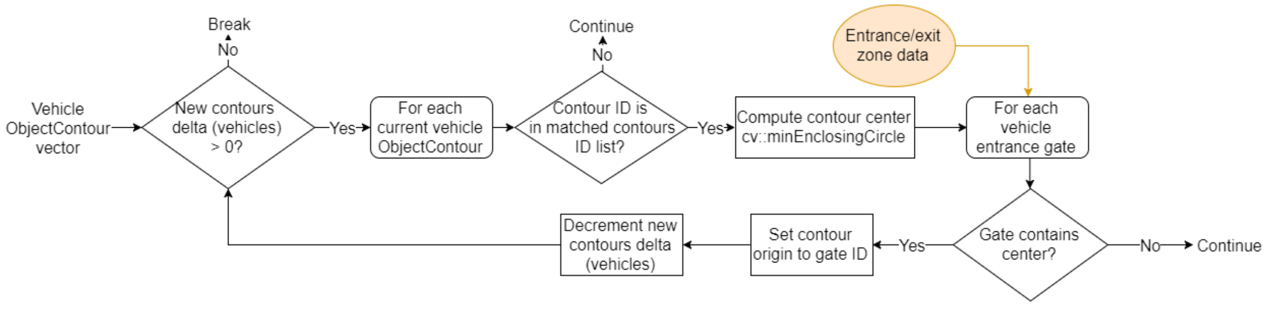

Break
No
Continue
No
Entrance/exit zone data
Vehicle ObjectContour vector
New contours delta (vehicles) > 0?
Yes
For each current vehicle ObjectContour
Contour ID is in matched contours ID list?
Yes
Compute contour center cv::minEnclosingCircle
For each vehicle entrance gate
Gate contains center?
No
Continue
Yes
Set contour origin to gate ID
Decrement new contours delta (vehicles)

Figure 39: Contour counting (identification) algorithm: setting new vehicle contour origin

Every contour in the current frame is checked for matches with an old contour (this is done by verifying if the contour's ID is in the `contour_id` vector (see section 3.5.3.2). If its ID is not present in that vector, it is a new contour and a candidate for a contour that just entered the frame. The position of this contour's center is then checked: if it is within one of the designated entrance zones, its origin parameter is set accordingly (see section 3.5.1) using `setOrigin()` and the number of new contours is decremented. If it reaches 0, the operation stops to avoid false positives.

If there are deleted contours (contours that disappeared), their point of exit must be determined. This approach is very similar to determining origin of new contours, but with a different set of conditions. The second part of the new/deleted contour identification algorithm is once again shown (**Figure 40**) and explained below.

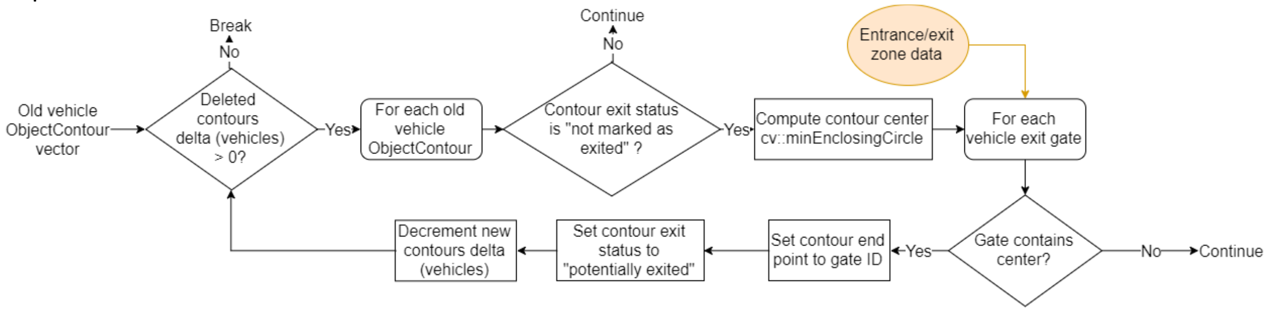
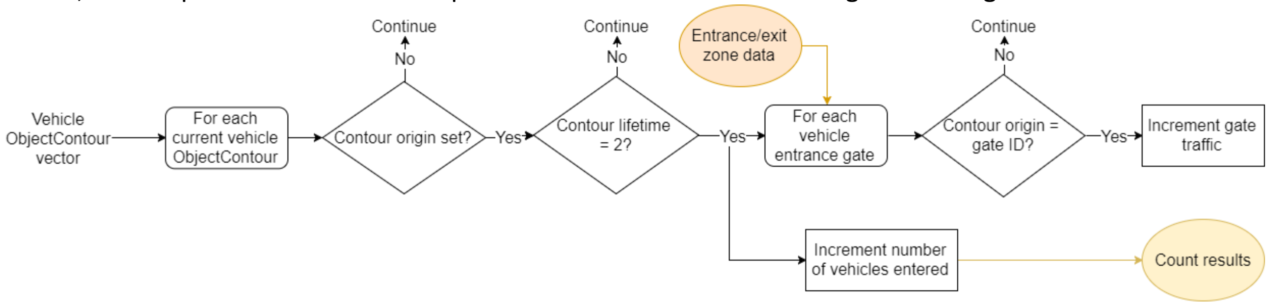

Break
No
Continue
No
Entrance/exit zone data
Old vehicle ObjectContour vector
Deleted contours delta (vehicles) > 0?
Yes
For each old vehicle ObjectContour
Contour exit status is "not marked as exited" ?
Yes
Compute contour center cv::minEnclosingCircle
For each vehicle exit gate
Gate contains center?
No
Continue
Yes
Set contour end point to gate ID
Set contour exit status to "potentially exited"
Decrement new contours delta (vehicles)

Figure 40: Contour counting (identification) algorithm: setting deleted vehicle contour end point

Every contour from the previous frame is checked for potential disappearance: the position of each contour's center is checked to determine if it is within one of the designated exit zones. If it is the case, an additional verification is performed to avoid glitches: the object's origin must not be the same as its exit point. If so, the object's exit status parameter is set to 0 (possibly exited) using `setExitStatus()` if it was not already set previously and its exit point is set using `setEnd()`. The number of deleted contours is then decremented. If it reaches 0, the operation stops to avoid false positives.

The next step is to confirm that new/deleted contours identified in the above algorithm are indeed valid events, and increment the appropriate counts (total vehicles passed and traffic through each gate). A new/deleted contour confirmation algorithm executes this function; its steps are shown in **Figure 41** below, then explained. Note that the procedure is different for entering and exiting vehicles.


Continue
No
Continue
No
Entrance/exit zone data
Continue
No
Vehicle ObjectContour vector
For each current vehicle ObjectContour
Contour origin set?
Yes
Contour lifetime = 2?
Yes
For each vehicle entrance gate
Contour origin = gate ID?
Yes
Increment gate traffic
Increment number of vehicles entered
Count results

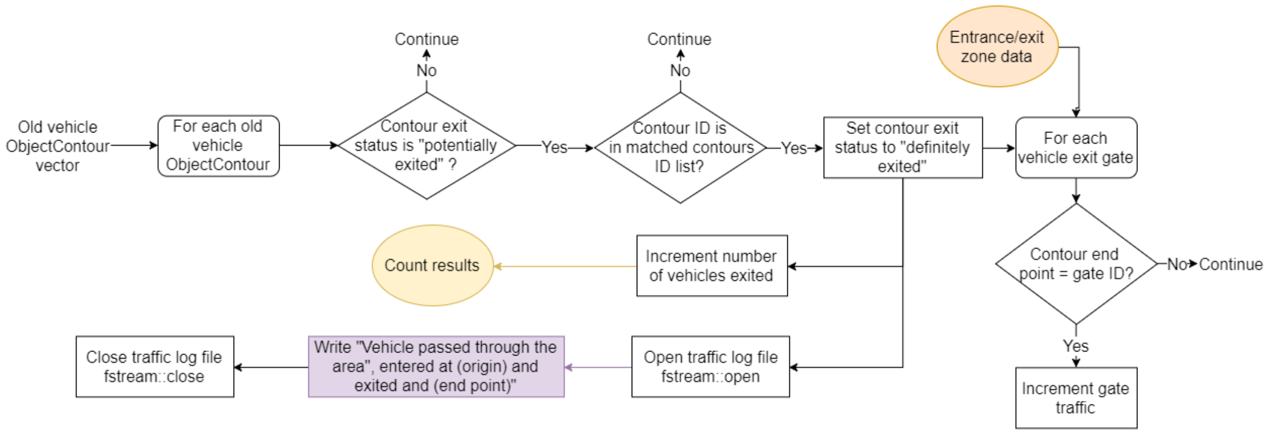

Old vehicle ObjectContour vector
For each old vehicle ObjectContour
Contour exit status is "potentially exited" ?
No
Continue
Yes
Contour ID is in matched contours ID list?
No
Continue
Yes
Set contour exit status to "definitely exited"
Entrance/exit zone data
For each vehicle exit gate
Contour end point = gate ID?
No
Continue
Yes
Increment gate traffic
Increment number of vehicles exited
Count results
Open traffic log file fstream::open
Write "Vehicle passed through the area", entered at (origin) and exited and (end point)"
Close traffic log file fstream::close

Figure 41: Contour counting (confirmation) algorithm for vehicles

For entering contours, after a vehicle has been assigned an origin via `setOrigin()`, the program waits for it to exist for two consecutive frames (lifetime equal to 2) to ensure it is an actual detection and not noise. Then, the number of vehicles entered is incremented, as well as the traffic count of the gate through which the contour entered.

For exiting contours, after a contour has been assigned an exit point via `setEnd()` and its exit status has been set to 0 via `setExitStatus()`, the program must verify that the vehicle has indeed exited the frame by checking if it is not present for two consecutive frames. Therefore, each contour in the second-to-last frame that has exit status 0 has its ID checked against all the IDs in `contour_id`. If it is not found, this means the vehicle has not been present for two consecutive frames and thus has most likely exited. In that case, the vehicle's exit status is set to 1 (certainly exited) via `setExitStatus()`, and every other contour with the same ID in all tracked frames also has its exit status set to 1 to avoid the same vehicle exiting being logged twice. Then, the total number of vehicles exited and traffic through the relevant exit zone is incremented. Also, whenever a vehicle has exited the frame, its trajectory (point of origin and exit) is logged to a log file with a timestamp (if the program lost track of the vehicle, its origin is marked as "unknown"). Unlike for new vehicles, a restriction on contour life time is not necessary.

3.5.3.3.2 Pedestrian counting

For pedestrian counting, the program must detect when and where pedestrians enter or exit the frame: through designated entrance and exit zones at the left, top, right, or bottom areas of the frame, but also outside of these areas (in the center of the frame). It also keeps track of the path of each individual pedestrian whenever possible. Unfortunately, the program can sometimes fail to keep track of a particular pedestrian if it stops in the middle of the frame or is obscured by another object. However, pedestrians appearing or disappearing outside of a designated zone do not affect the count of pedestrians entering or leaving the frame, so it is not a major issue. The algorithm also has identification and confirmation steps, like vehicle counting (see section 3.5.3.3.1).

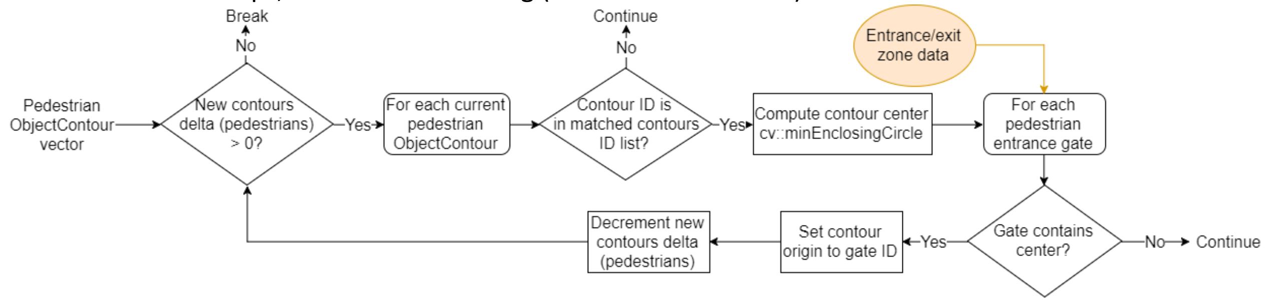
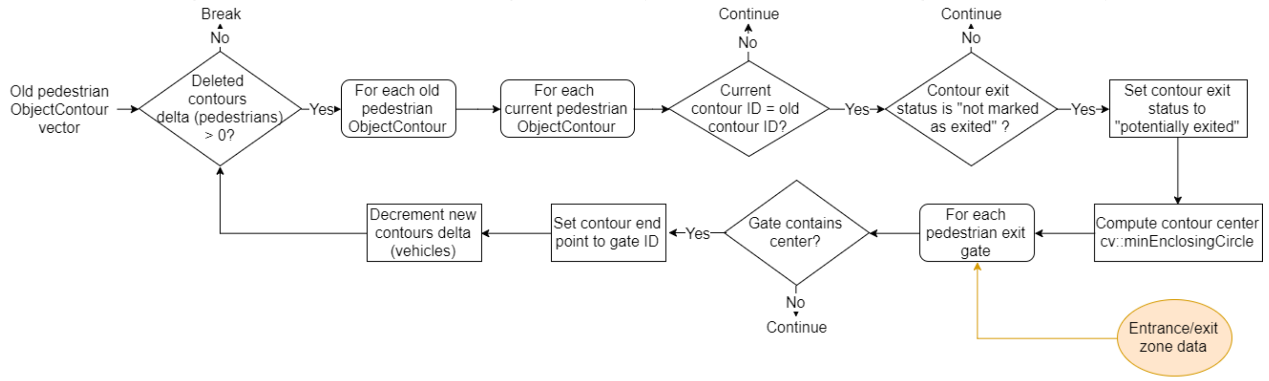

Pedestrian ObjectContour vector
New contours delta (pedestrians) > 0?
No
Break
Yes
For each current pedestrian ObjectContour
Contour ID is in matched contours ID list?
No
Continue
Yes
Compute contour center cv::minEnclosingCircle
Entrance/exit zone data
For each pedestrian entrance gate
Gate contains center?
No
Continue
Yes
Set contour origin to gate ID
Decrement new contours delta (pedestrians)

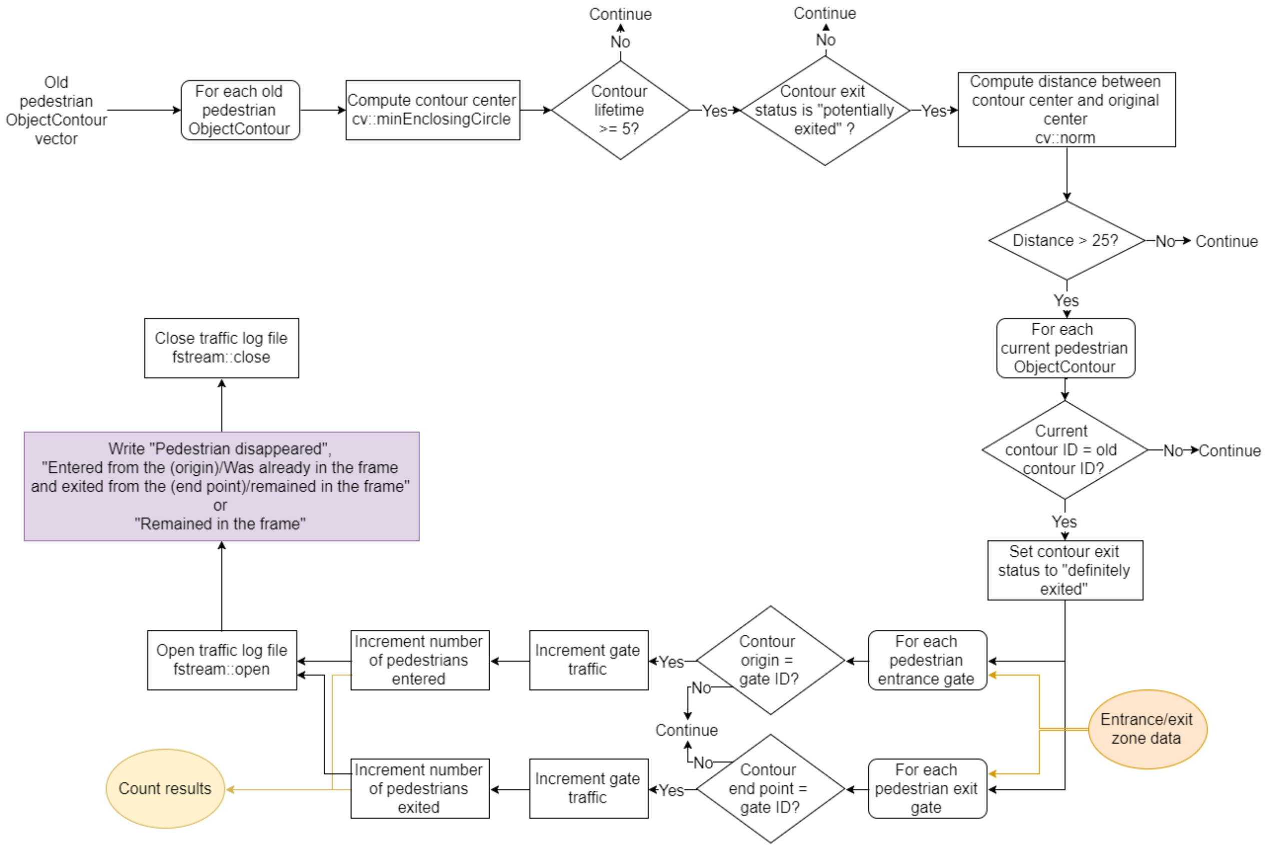
Figure 42: Contour counting (identification) algorithm: setting new pedestrian contour origin

Similarly to vehicle detection, if there are new contours, their point of origin must be determined by checking for matches with an old contour (see section 3.5.3.3.1). This is done using the contour entering/exiting identification algorithm in **Figure 42** below, which is explained further down.
For new contours, the position of this contour's center is then checked: if it is within one of the designated entrance zones, its origin parameter is set accordingly (see section 3.5.1) using `setOrigin()` and the number of new contours is decremented. If it reaches 0, the operation stops to avoid false positives.
If there are deleted contours, their point of exit must be determined by checking for potential disappearance. However, unlike for vehicles, the point of exit can be anywhere inside the frame, so the identification algorithm is different. The algorithm steps are shown below (**Figure 43**) and explained.



Break
No
Continue
No
Continue
No
Old pedestrian ObjectContour vector
Deleted contours delta (pedestrians) > 0?
Yes
For each old pedestrian ObjectContour
For each current pedestrian ObjectContour
Current contour ID = old contour ID?
Yes
Contour exit status is "not marked as exited" ?
Yes
Set contour exit status to "potentially exited"
Decrement new contours delta (vehicles)
Set contour end point to gate ID
Yes
Gate contains center?
For each pedestrian exit gate
Compute contour center cv::minEnclosingCircle
No
Continue
Entrance/exit zone data

Figure 43: Contour counting (identification) algorithm: setting deleted pedestrian contour end point



Continue
No
Continue
No
Old pedestrian ObjectContour vector
For each old pedestrian ObjectContour
Compute contour center cv::minEnclosingCircle
Contour lifetime >= 5?
Yes
Contour exit status is "potentially exited" ?
Yes
Compute distance between contour center and original center cv::norm
Distance > 25?
No
Continue
Yes
For each current pedestrian ObjectContour
Current contour ID = old contour ID?
No
Continue
Yes
Set contour exit status to "definitely exited"
Close traffic log file fstream::close
Write "Pedestrian disappeared", "Entered from the (origin)/Was already in the frame and exited from the (end point)/remained in the frame" or "Remained in the frame"
Open traffic log file fstream::open
Increment number of pedestrians entered
Increment gate traffic
Yes
Contour origin = gate ID?
For each pedestrian entrance gate
No
Continue
No
Entrance/exit zone data
Count results
Increment number of pedestrians exited
Increment gate traffic
Yes
Contour end point = gate ID?
For each pedestrian exit gate

Figure 44: Contour counting (confirmation) algorithm for pedestrians

Checking for contours inside designated exit zones, like for vehicles, is not enough. Therefore, for pedestrians, checking if a particular contour has disappeared involves a process similar to the one in `calculate_diff()` function: the ID of each contour from the previous frame is compared to that of all current frame contours. If a matching contour is not found and the old contour's life time is greater or equal to 5 frames (to avoid glitches), the contour's life time is set to 0 (possibly exited) using `setExitStatus()` and its exit point is set accordingly using `setEnd()` (if the contour's origin and exit point are the same, the exit point is set to -2, a special value for this purpose). The number of deleted contours is then decremented. If it reaches 0, the operation stops to avoid false positives.

The new/deleted contour confirmation algorithm is vastly different from that of vehicles. Due to the different approach used for identifying entering/exiting contours in the above algorithm, the confirmation algorithm can be simplified: it only needs to consider pedestrian exiting events. The algorithm is shown (**Figure 44**) and explained below.

For pedestrian counting, the program ignores events corresponding to pedestrians appearing the frame, instead doing all the counting when a pedestrian exits the frame. This is impossible for vehicles because it does not track vehicles that disappear in the middle of the frame, so they must be counted when they enter or information is lost. However, for pedestrians, the software generates an event at any time a pedestrian exits, regardless of whether it is in a designated exit zone, so all the events can be generated upon disappearance of a pedestrian.

After a pedestrian contour has been assigned an exit point via `setEnd()` and its exit status has been set to 0 via `setExitStatus()`, the program must verify that the pedestrian has indeed exited the frame by checking if it is not present for two consecutive frames. Every contour whose exit status has been set to 0 is checked for two conditions to remove noise: it must have a life time greater than 5 frames, and have traveled a minimum of 25 pixels to be considered a valid pedestrian. If this is the case, its ID is once again checked against all IDs of current frame contours, and if a match is not found, the contour has definitely exited the frame. Its exit status and that of every contour with the same ID is set to 1 (certainly exited) using `setExitStatus()`). Its exit point is reset to -1 (default, not in any designated zone) and checked again, since it might have varied. Numbers from 0 to 3 indicate a valid exit zone, -2 indicates that the contour has entered and exited from the same point and is not to be counted, since it is very likely a glitch. Then, based on the origin set previously and the exit point, an appropriate message is generated and saved to a log file, and the pedestrian count in the frame is changed. A pedestrian that has entered and exited through (different) designated zones will generate a "Entered from the *location* and exited from the *location*" message and not change the count. A pedestrian that has entered through a designated zone and disappeared inside the frame will generate a "Entered from the *location* and stayed in the frame" message and increase the count by 1. A pedestrian that has appear inside the frame and has exited through a designated zone will generate a "Was already in the frame and exited from the *location*" message and decrease the count by 1. A pedestrian that has appeared and disappeared inside the frame will generate a "Remained in the frame" message and not change the count. This information is written to the log file along with an event time stamp, so the code can easily be adapted to interact with a proper database.

3.5.3.3.3 Database logging

The gate status algorithm, the motion detection software's database logging algorithm, is very simple, since it does not interpret data given to it further unlike the parking detection logging algorithm (see section 3.3.1). The steps are shown in **Figure 45** below and are self-explanatory.

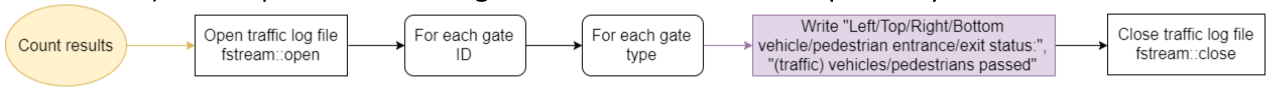

Count results
Open traffic log file fstream::open
For each gate ID
For each gate type
Write "Left/Top/Right/Bottom vehicle/pedestrian entrance/exit status:", "(traffic) vehicles/pedestrians passed"
Close traffic log file fstream::close

Figure 45: Motion detection database logging algorithm

3.5.4 Dead zones

One particularity of the object counting algorithm developed in this project is that it requires contours to appear or disappear to be counted as an object that entered or exited, instead of simply relying on the object's contour crossing the region of interest. This method of contour detection is very robust because

it is much less likely to result in an object being counted multiple times, unlike in competing software (refer to section 4.2 for results). However, the main disadvantage of this method is that it limits flexibility of camera placement: it must be configured such that the entrance and exit zones are placed exactly at the edge of the frame. Furthermore, it is sometimes far too difficult to detect individual objects at the edge of the frame, let alone count them because they appear too small.
The solution was to introduce dead zones, or regions of the frame where no motion will be detected: in the difference frame, the regions corresponding to dead zones will be blacked out. They are declared very similarly to entrance or exit zones (see section 3.1.1). This solves the aforementioned issue: if a dead zone is placed between an entrance zone and the edge of the frame, it ensures that contours entering the frame will only be detected once they appear in the entrance zone. Similarly, if the dead zone is between an exit zone and the edge of the frame, any contour that passes through the exit zone will disappear shortly after. For an example of how dead zones can enable entrance or exit zones non-adjacent to the frame edges, see **Figure 46**:

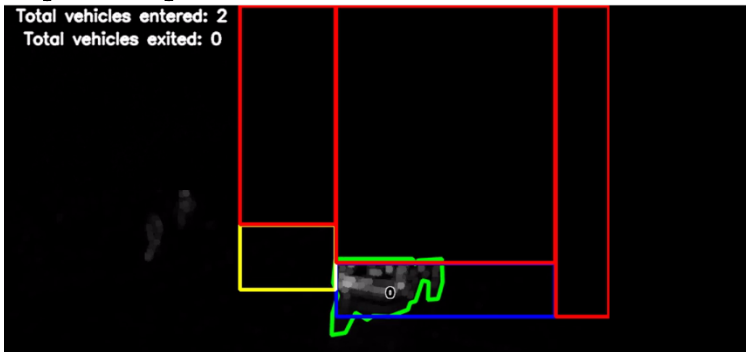


Figure 46: Vehicle about to disappear through gate away from edge of frame thanks to dead zone

Another advantage of dead zones is that they allow masking out some regions of the frame where undesired contours may appear. For example, if one wishes to count vehicles traveling across a road, it is possible that cyclists passing on the nearby bike path interfere with the count: if a car and a group of bikes are detected as a single contour, its centroid may no longer be within the entrance zone designated for the car, and the count will be missed. Placing a dead zone over the bike path will ensure that the cyclists will be masked out and will not interfere with the proper detection of the vehicle. For an example of improvements to vehicle contour detection due to motion detection dead zones, see Figure 47:

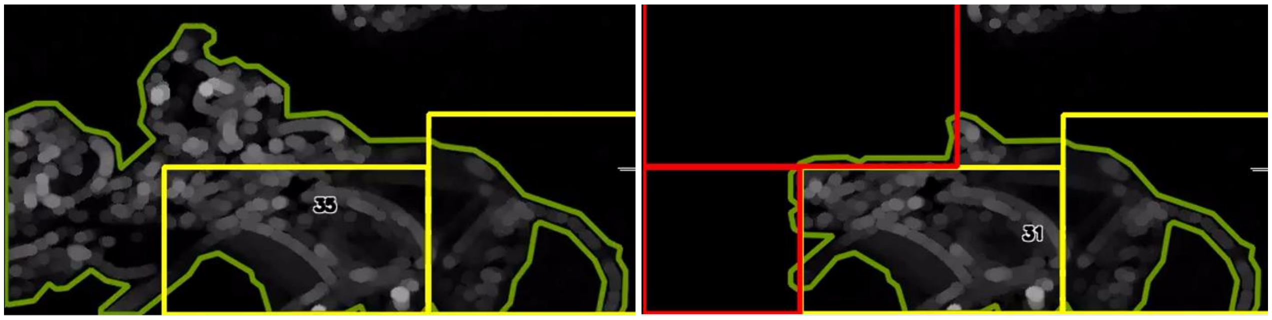


Figure 47: Comparison of contour detection with deadzones disabled (poor contour detection) and enabled (better contour detection)

3.5.5 Defunct features

3.5.5.1 Background subtraction for motion detection

In computer vision motion detection, there are two main techniques to identify moving objects: the one used in this project, computing the absolute difference between two (or three) consecutive frames, and separating foreground from background. The two techniques are compared in [36] and the author determines that subtracting consecutive frames has a significant disadvantage: if the object is moving slowly, it will not be detected as a whole contiguous object but as multiple small objects, and could even not be detected at all. Instead, each frame can be compared with the first frame of the video sequence, but this also has disadvantages: if the first frame contains a static object that then leaves the frame or if a new object permanently enters the frame and stays there, motion will be detected continuously [36]. The best algorithms are based on a variable background of the scene, with various algorithms to build the background that are "too complex" [36]. Fortunately, OpenCV has many such built-in algorithms.

One algorithm that was strongly considered using is the `BackgroundSubtractorMOG2`, implemented as `cv::BackgroundSubtractorMOG2` and discussed in section 3.4.2. This algorithm is very easy to use: a single `apply()` statement takes the current frame and outputs the foreground as a binary mask, and the background is built automatically without the user's intervention. Unfortunately, this method did not succeed: it was too sensitive to motion, classifying even very small moving objects as part of the foreground. Furthermore, the same issue as the static background example in [36] occurred: static objects were integrated in the background and when they exited, false motion was continuously detected for several seconds until the background was updated. The subtraction of consecutive frames was thus chosen, an approach successfully executed in [8] [9] [10]. To solve the lack of sensitivity to small motion, dilation and thresholding were used to make moving objects much more apparent (see section 3.4.3). This method worked much better than the complex `BackgroundSubtractorMOG2`, so that idea in favor of an algorithm that was also much simpler and faster.

3.5.5.2 Contour separation algorithm

One issue that occurred in the object detection portion of the software was that nearby objects were detected as a single, large object by the `cv::findContours()` function because their shapes overlapped after dilation and thresholding was applied. Since very little control can be exerted over the sensitivity of the `findContours()` function, the difference frame had to be modified. The initial idea for a solution was a contour separation algorithm: after contours were found using `findContours()` and irrelevant contours were eliminated, erosion would be applied on the resulting frame. This would have the effect of separating objects that were connected by thin "bridges", since erosion has the effect of reducing the width of white objects on a dark background. Each of the detected contours would then be copied to its own frame, which would each be dilated, then `findContours()` would be applied to each of them and the contours would be detected individually. This would restore each original contour's size without causing the contours to merge again, allowing two distinct contours that overlap slightly.

Unfortunately, this technique was not successful: an optimal erosion kernel size that was successful in separating all contours while not completely eliminating some could not be found. It was also too difficult to monitor and debug this algorithm. It was abandoned it in favor of separating the frame into a "top" and "bottom" region, to which different dilation kernels and minimum thresholds for the thresholding operation. This approach makes sense: after all, objects at the top of the frame are smaller than those at the bottom, so should be affected by a proportionally smaller kernel, but implementing a smooth kernel size transition would be too complex, so a rudimentary approximation is sufficient. This new approach ended up working much better than the very complex and slow separation algorithm, since the original issue of nearby objects clustering occurred mostly at the top of the frame.

3.5.5.3 Shadow mask with CIE L*a*b*/Cie L*u*v* color space

As mentioned in section 3.4.2, the shadow mask generation algorithm relies on the HSV color space (sometimes referred to as HSB). Although it is simple to understand (hue corresponds the type of color, saturation corresponds to the strength of color, value or brightness corresponds to the brightness), it is not optimal for several reasons and "HSB and HLS should be abandoned" [37]. The main reason is that HSV is not perceptually uniform: a small change to one of the component values is not approximately equally perceptible across the entire range of the value, unlike the CIE L*a*b* and CIE L*u*v color spaces [37]. For example, a brightness value of 50% should always appear to be half as bright as a value of 100%, but yellow is approximately six times more intense than blue at the same brightness due to a simplified, inadequate formula [37]. Furthermore, the hue is defined as an angular value (from 0 to 360°), which makes computations difficult [37]. Even worse, hue is piecewise-defined (different formulas for each 60° portion of the circle), which leads to visible discontinuities in the hue channel [37]. Regardless, the HSV color space was kept instead of transitioning to CIE L*a*b* or CIE L*u*v. The main reason is the complexity of the formulas for converting RGB to either of these color spaces, which involve several steps and are the following [38]:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{13}$$

$$L = \begin{cases} 116 \cdot Y^{1/3}, Y > 0.008856 \\ 903.3 \cdot Y, Y \leq 0.008856 \end{cases} \tag{14}$$

$$u' = 4 \cdot \frac{X}{X + 15 \cdot Y + 3 \cdot Z} \tag{15}$$

$$v' = 9 \cdot \frac{Y}{X + 15 \cdot Y + 3 \cdot Z} \tag{16}$$

$$u = 13 \cdot L \cdot (u' - 0.19793943) \tag{17}$$

$$v = 13 \cdot L \cdot (v' - 0.46831096) \tag{18}$$

$$f(t) = \begin{cases} t^{1/3}, t > 0.008856 \\ 7.787t + \frac{16}{116}, t < 0.008856 \end{cases} \tag{19}$$

$$a = 500 \cdot \left(f(X) - f(Y)\right) + 128 \tag{20}$$

$$b = 200 \cdot \left(f(Y) - f(Z)\right) + 128 \tag{21}$$

From these equations, one can see that converting to and from these color spaces is somewhat computationally intensive. *Formula 13* is a simple linear transformation, but *Formulas 14, 19* involve exponentiation operations and *Formulas 15, 16* involve floating-point division. Despite this, using either the CIE L*a*b* or CIE L*u*v color space did not improve the results of shadow mask generation. Indeed, the main issue with shadow detection is that surfaces of black vehicles are very similar to shadows at the pixel level: the ratio of foreground brightness to background brightness changes by approximately the same amount and the color changes very little from foreground to background. Therefore, regardless of the color space used, it is extremely difficult to distinguish shadows from black vehicles, and it is not worth using a more computationally intensive, less intuitive color space.

3.5.5.4 Matching contours to known shapes

For vehicle detection, it is of utmost importance to detect and filter out contours corresponding to pedestrians and cyclists. In ideal conditions (daytime), cyclists and pedestrians are too small to be detected, but in the morning and evening, the shadows they cast make the entire contour exceed the vehicle size threshold (see section 3.5.3.1 for more details). Therefore, the implementation of a shape-based filter implementation that would remove contours whose shape resembles a reference contour was attempted. This is achieved using the `cv::matchShapes()` function, which determines how closely two shapes are related using the Hu invariants: a set of seven image moments that are invariant to image scale, rotation, and reflection [33]. This property of the Hu invariants is extremely useful because the reference image may be found in many different sizes and orientations, and may even be reflected (in the evening, when the shadow faces the other way). For more information on the Hu invariants and their definitions, refer to [39]. The `cv::matchShapes()` function returns a positive floating-point value that is smaller the more the two images are similar to each other, so a threshold was calibrated, below which a given contour is close enough to the typical shape of a pedestrian and its shadow. The reference contour used in this case is shown in **Figure 48**:

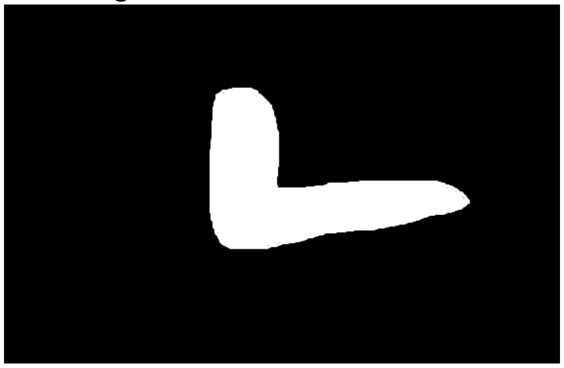


Figure 48: Reference image of a pedestrian and its shadow

Unfortunately, this method did not work as well as expected. The main issue was that when multiple contours belonging to pedestrians or cyclists are near each other, they can overlap and the resulting contour looks nothing like the reference contour. Ultimately, a more basic, yet more functional solution is to use the convexity of the shape to eliminate the undesirable contours (see section 3.5.3.1).

4. Results, comparisons, limitations

4.1 Parking detection results

4.1.1 Performance analysis

Table 2: Parking detection results (using software developed in this project)

Stream name	Stream info	Number of parking spots	Number of mistakes	Percentage accuracy
1	172.168.10.30, 05 May 2017 09:57:39-09:58:02	11	0	100%
2	172.168.10.30, 05 May 2017 10:10:22-10:10:49	11	0	100%
3	172.168.10.30, 05 May 2017 10:38:16-10:38:36	11	0	100%
4	172.168.10.30, 05 May 2017 10:43:57-10:44:26	11	1	90.9%
5	172.168.10.30, 05 May 2017 11:21:37-11:22:16	11	0	100%
6	172.168.10.30, 05 May 2017 11:32:57-11:33:06	11	0	100%
7	172.168.10.30, 05 May 2017 11:55:11-11:55:27	11	0	100%
8	172.168.10.30, 05 May 2017 11:57:37-11:58:49	11	1	90.9%
9	172.168.10.30, 05 May 2017 12:49:22-12:49:50	11	0	100%
10	172.168.10.30, 05 May 2017 12:52:52-12:53:19	11	0	100%
11	172.168.10.30, 05 May 2017 13:01:57-13:02:43	11	0	100%
12	172.168.10.30, 05 May 2017 13:39:48-13:40:52	11	0	100%
13	172.168.10.30, 05 May 2017 14:04:18-14:04:57	11	0	100%
14	172.168.10.30, 05 May 2017 14:43:14-14:43:43	11	1	90.9%
15	Clip included with original software [6], duration 02:29	16	0	100%
Total		170	3	98.2%

With all the above modifications to the parking detection logic and image processing, the reliability of the parking detection software in typical conditions (from early morning to late evening, with good weather conditions) is more than satisfactory. Testing footage consisted of 14 short clips of an intersection in downtown Montreal with 11 parking spots each, as well as a test video file included with the original software [6] which features 16 parking spots, for a total of 170 parking spots monitored. The test file does not pose much of a challenge, since it is a parking lot and there is little activity other than cars entering or leaving parking spots. Also, the camera angle is such that a car in a parking spot only covers a small portion of adjacent parking spots. On the contrary, the downtown intersection is much more challenging: cars cover adjacent parking spots due to the poor camera angle, some parking spots are only partially visible, and there is a lot more activity (for example, traffic). The performance of the parking software is defined based on the number of mistakes: detections that would cause an uncertain or wrong parking status detection and last more than approximately a second, and is quantified according to the following formula (*Formula 22*):

$$\left(1 - \frac{Number\ of\ mistakes}{Number\ of\ parking\ spots}\right) \cdot 100\% \qquad (22)$$

While viewing the footage, only 3 mistakes were recorded which corresponds to an accuracy of 167/170, or 98.2% (for detailed data, see Table 2 below). These mistakes mostly occurred on a single parking spot, so the reliability could be improved significantly by tweaking the parameters related to that spot only. As expected, the performance of the software is relatively consistent, with only 0 or 1 mistakes in each clip. Also, as expected, the errors all occurred in the downtown intersection, not the significantly easier parking lot test clip. The following table shows detailed results:

4.1.2 Non-ideal conditions

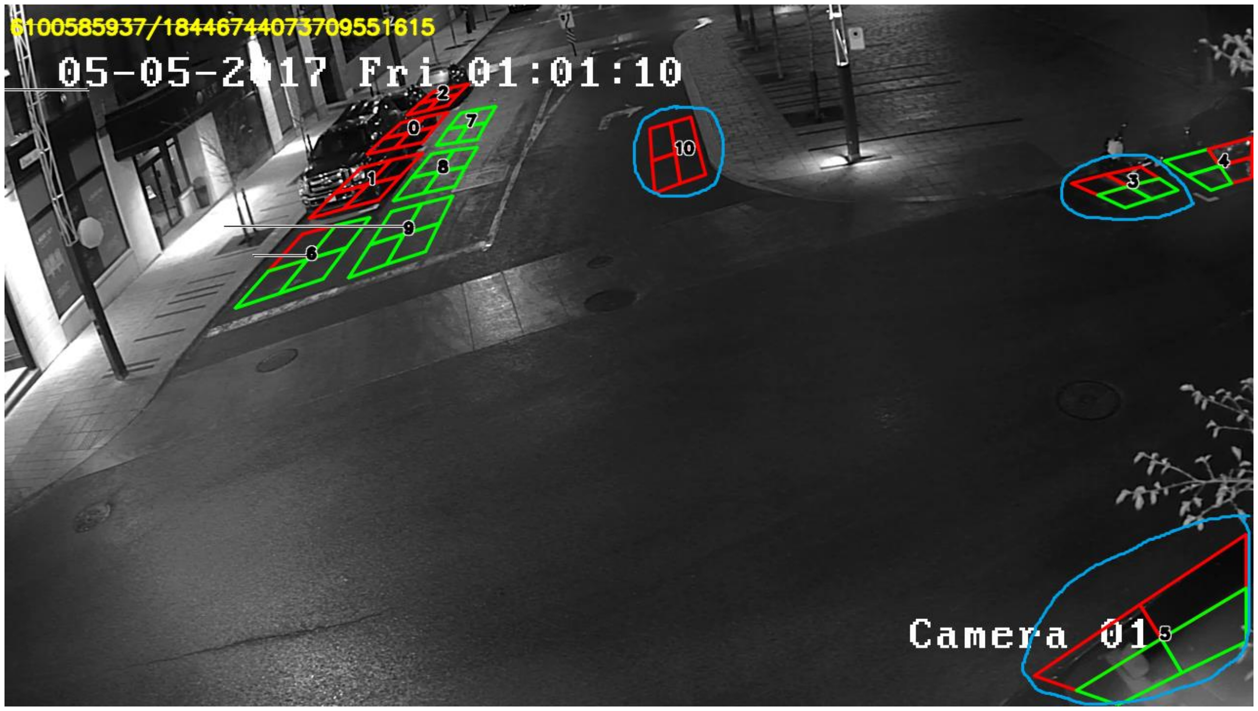


Figure 49: Parking detection issues during the night

Unfortunately, in less ideal conditions (in the middle of the night, during rain or wind), the software does not perform as reliably. During the night, most of the frame is too dark, which makes it very difficult to detect edges: the edge-based algorithm detects much less vehicles than in the daytime, leaving a significant portion of the work to the luminance-based algorithm, which is inherently less reliable. Also, the lighting of the frame is not uniform due to bright spots created by street lamps and dark spots where no lamp happens to shine. While histogram equalization via CLAHE (see section 3.2.2) helps make the average luminance comparable to that of the day, it cannot deal with the uneven lighting effectively, leaving some areas overexposed or underexposed despite its adaptive nature that promises to prevent this issue. This leads to false detections by the luminance-based algorithm. The following image illustrates these issues:

Figure 49 demonstrates that the edge-based detection algorithm functions poorly at night: in parking spot 5, the lower two quadrants are not marked as occupied despite the presence of a vehicle with quite a few edges. It also shows the shortcomings of CLAHE: illegal parking zone 10 is marked as occupied even though there is nothing there, since it is a very dark region (one of the darkest in the whole frame).

Another weather condition that causes issues is the rain. As mentioned in section 3.2.3, it causes the texture of the asphalt to become too rough, which can be solved with denoising. However, doing so incurs a significant performance penalty, even with optimizations. Moreover, rain makes the asphalt reflective. This poses a problem for the edge-based detection algorithm, which detects the reflections of objects as objects (false detections). This can occasionally cause errors in the detection algorithm, as illustrated in **Figure 50**. It demonstrates that the edge-based algorithm can sometimes detect reflections. In particular, the entirety of parking spot 3 is shown as occupied even though the scooter occupies only half of the area; the other half is covered by its reflection. This is not a problem in the current setup, where parking spot 3 is a single zone, but if it was split into two zones (for two scooters), an empty spot would be marked as occupied.

The last problem related to operating conditions is wind, which causes the camera mount to shake and occasionally lose focus, making the image blurrier for a few seconds while the camera recovers and re-focuses. The blur leads to a very low value reported by edge detection algorithm, making it difficult to identify if a spot is occupied (a similar effect as that described in section 3.2.1). Since this effect is typically short-lived, it does not cause any issues. However, on a very windy day, it could very well persist longer than a few seconds. **Figure 51** demonstrates the sensitivity of the edge detection algorithm to blur caused by wind: the vehicles in parking zones 0, 2, and 6 are not being detected properly. This is only an issue if the loss of focus coincides with the moment when the program samples the status of each parking spot for logging it to file (it will lead to one wrong set of parking statuses being written, which will be rectified at the next event log). However, in an excessively windy day, this error could occur multiple times.

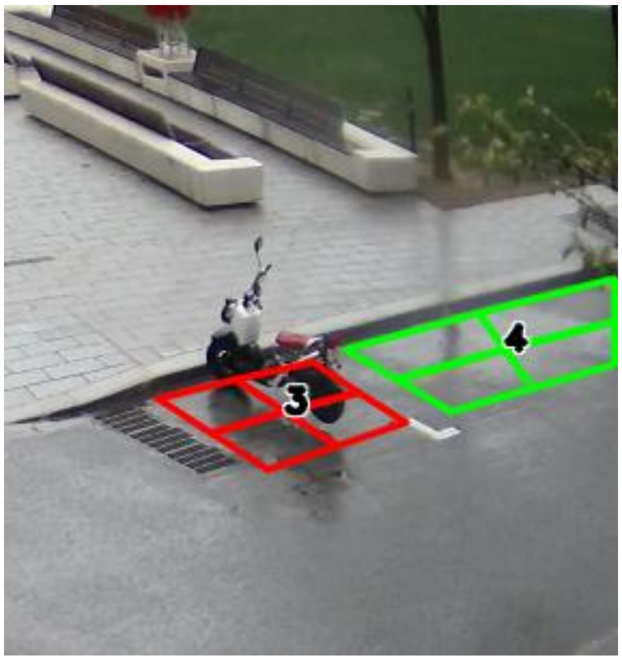


Figure 50: Parking detection issues during rainy weather (zone 3)

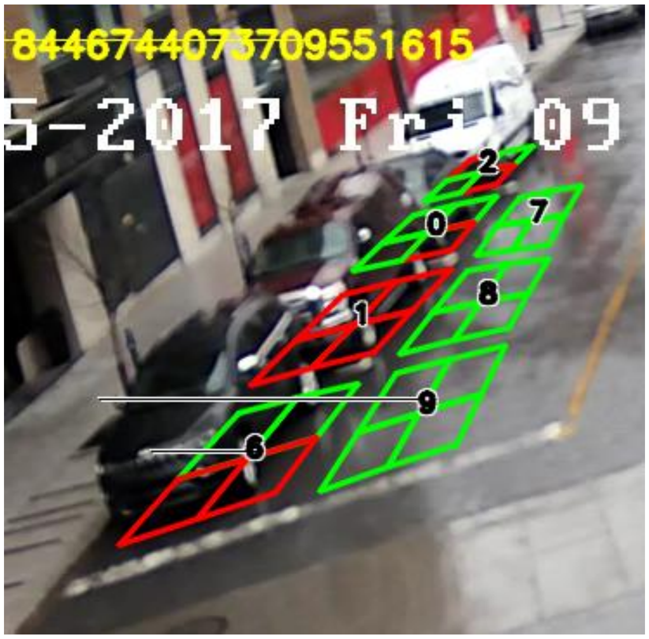


Figure 51: Parking detection issues during wind (zones 0, 2, and 6)

4.1.3 Comparison to other software

Table 3: Parking detection results (using reference software [6])

Stream name	Stream info	Number of parking spots	Number of mistakes	Percentage accuracy
1	172.168.10.30, 05 May 2017 09:57:39-09:58:02	11	2	81.8%
2	172.168.10.30, 05 May 2017 10:10:22-10:10:49	11	4	63.6%
3	172.168.10.30, 05 May 2017 10:38:16-10:38:36	11	4	63.6%
4	172.168.10.30, 05 May 2017 10:43:57-10:44:26	11	2	81.8%
5	172.168.10.30, 05 May 2017 11:21:37-11:22:16	11	0	100%
6	172.168.10.30, 05 May 2017 11:32:57-11:33:06	11	0	100%
7	172.168.10.30, 05 May 2017 11:55:11-11:55:27	11	0	100%
8	172.168.10.30, 05 May 2017 11:57:37-11:58:49	11	1	90.9%
9	172.168.10.30, 05 May 2017 12:49:22-12:49:50	11	0	100%
10	172.168.10.30, 05 May 2017 12:52:52-12:53:19	11	1	90.9%
11	172.168.10.30, 05 May 2017 13:01:57-13:02:43	11	2	81.8%
12	172.168.10.30, 05 May 2017 13:39:48-13:40:52	11	2	81.8%
13	172.168.10.30, 05 May 2017 14:04:18-14:04:57	11	2	81.8%
14	172.168.10.30, 05 May 2017 14:43:14-14:43:43	11	1	81.8%
15	Clip included with original software [6], duration 02:29	16	1	93.8%
Total		170	22	87.1%

To quantify the impact of the performance improvements made to the parking detection software, the results of the modified version and the original one were compared using the same video clips. The unique

Laplacian threshold parameter was fine-tuned until the best possible results were obtained, which are shown in Table 3 below:
The initial software performs decently enough, with an overall count of 22 mistakes and overall accuracy of 87.1%, considering how limited the options for fine-tuning are. However, it does not even come close to the 98.2% achieved by the modified version of the software with semi-optimized settings; it would be possible to increase this even further. In addition to this, for each test, the modified version of the software has better performance, so no feature that was added ended up worsening overall performance. It should also be noted that the original software does not have any provisions for dealing with difficult conditions such as night-time, so it may work very poorly or not at all at night.

4.2 Motion detection results

4.2.1 Consistency and accuracy analysis

To test the motion detection software, three five-minute clips from the Jeanne-Mance and President-Kennedy intersection were selected for vehicle detection, and one ten-minute clip from the Clarke and Ontario intersection for pedestrian detection, all of which were in typical conditions (morning and day, with good weather conditions). The test clip included with the reference software [34] that was tested in section 4.2.3 was also used, which is not very challenging compared to the application in this project (top-down view, contains only vehicles, only a few seconds long).
To quantify the performance of the motion detection software, two metrics were used: consistency, and accuracy. In this case, consistency refers to the consistency between the recorded count of objects that have entered the frame and the count of those that have exited (the values are adjusted based on the initial and final number of vehicles in the frame). To calculate the performance of the software in the consistency category, *Formula 23* is used (the percentage error between the count of objects that entered and exited is subtracted from 100%):

$$\left(1-\frac{|Objects\ entered{:}count-Objects\ exited{:}count|}{\frac{1}{2}\cdot(Objects\ entered{:}count+Objects\ exited{:}count)}\right)\cdot 100\% \qquad (23)$$

As for accuracy, it refers to the closeness of the count of objects entered/exited to the actual number of objects entered/exited. Accuracy values are computed separately for both objects entering and objects exiting since the algorithms used are slightly different and can be adjusted independently to improve the performance of one without affecting the other. To calculate the accuracy, *Formula 24* is used (the percentage error between the count and actual value of objects that entered/exited is subtracted from 100%):

$$\left(1-\frac{|Objects\ entered{,}exited{:}actual-Objects\ entered{,}exited{:}count|}{Objects\ entered{,}exited{:}actual}\right)\cdot 100\% \qquad (24)$$

The performance values for a few stream segments are displayed in the following table (Table 4):
Table 4: Motion detection results (using software developed in this project)

Stream name	Stream info	Obj. enter (cnt.)	Obj. enter (act.)	Obj. exit (cnt.)	Obj. exit (act.)	Consistency	Accuracy (enter)	Accuracy (exit)
Example 1 (vehicles)	172.168.10.32 14 June 2017 08:55-09:00	55	52	47	50	88%	94.23%	94%
Example 2 (vehicles)	172.168.10.32 14 June 2017 12:10-12:15	53	50	51	52	92.16%	94%	98.08%
Example 3 (vehicles)	172.168.10.32 14 June 2017 12:30-12:35	58	60	53	56	98.13%	96.67%	94.64%

Example 4 (pedestrians)	172.168.10.30 05 May 2017 12:00-12:10	20	22	18	20	100%	90.91%	90%
Example 5	Clip included with reference software [34], duration 00:12	5	5	5	5	100%	100%	100%

*Note: The software was found to perform worse (both in terms of consistency and accuracy when the shadow mask was enabled, so this algorithm was disabled when collecting this data

From the above performance values, one can see that the performance of the motion detection software is excellent, for both pedestrians and vehicles. In all cases, both consistency and accuracy are above 90% or very near that figure, so the error introduced over the true measurement is under 10%. Most of the time, the count of vehicles entering is higher than the actual number of vehicles that passed through the area, which is most likely due to vehicles entering a designated entrance zone, stopping within, and then starting to move again, which leads to a double count. As for the vehicle exit counts, they are systematically lower than the actual vehicle counts. One explanation for this lack of detection is that vehicles are moving particularly slowly during that period, so motion detection fails to detect some vehicle (insufficient motion to exceed threshold). Another reason for the poor result is that vehicles are close to each other: object detection fails to distinguish these vehicles and groups them as one object, a phenomenon that is exacerbated in the morning because the shadow mask is currently disabled.

Considering the above results, the motion detection software works very well, but can be improved further. The most obvious change would be to improve contour segmentation to ensure better detection of multiple nearby vehicles. This would heavily involve improvements to the shadow masking algorithm (see section 5.2.3.1), as well as a potential decrease in dilation kernel size (see section 3.4.3). Another one would be to increase sensitivity to low motion, which causes objects to be barely visible in the difference frame; this would mainly be achieved through a decrease in white threshold (see section 3.4.3).

4.2.2 Non-ideal conditions

4.2.2.1 Long shadows

The motivation for developing the shadow mask (see section 3.4.2) were the long shadows that objects cast early in the morning or in the evening, which are detected as moving objects. They often cause multiple actual objects to be detected as a single one if the shadow of a given object overlaps with another. However, a disadvantage of the shadow mask is that dark portions of vehicles (namely windows and the bodies of black and grey vehicles) are extremely similar to shadows, so they can be falsely detected and masked. This implies a tradeoff in using the shadow mask: it can either be very effective at removing shadows and cause misses of dark vehicles in the process, or less effective at removing shadows but safer against missing actual vehicles. Setting the parameters of the shadow mask appropriately is thus a very delicate balancing act, and in the end it was determined that the best results (see section 4.2) would be obtained by disabling it. As expected, in this configuration, the highest percentage errors occur during the morning test scene (Example 1: 88% consistency, 94.23%/94% accuracy), where shadows caused some detection mistakes, while the daytime scenes (Example 2: 92.16% consistency, 94%/98.08% accuracy and Example 3: 98.13% consistency, 96.67%/94.64% accuracy) show much better results. Therefore, long shadows definitely pose a problem in the current optimal configuration (where the shadow mask is not active).

For examples of advantages and disadvantages of the shadow mask when enabled or disabled, see the following (**Figure 52**, **Figure 53**):

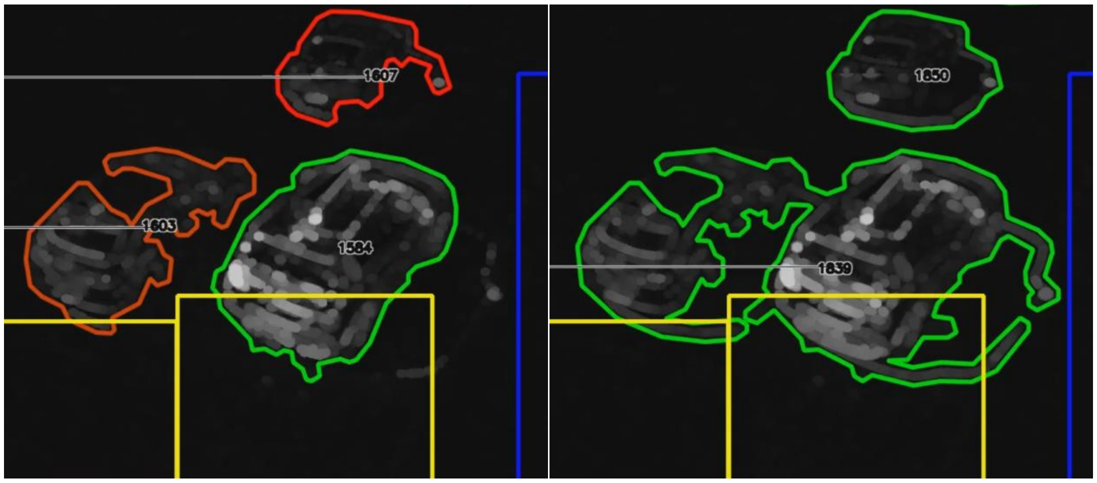


Figure 52: Shadow mask advantage: better ability to distinguish multiple vehicles when shadow mask on (left) than off (right)

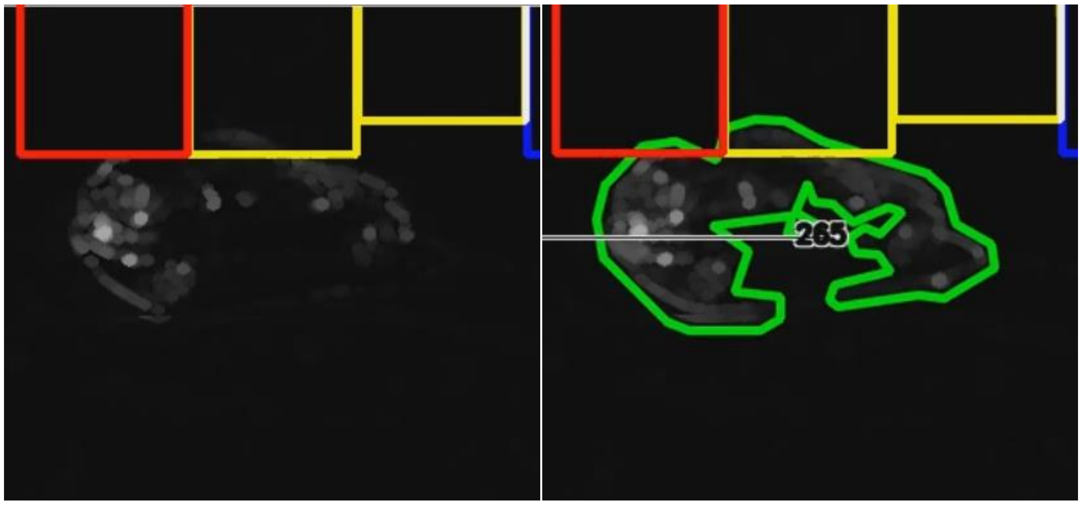


Figure 53: Shadow mask disadvantage: detection misses of dark vehicles when shadow mask on (left) and not when off (right)

4.2.2.2 Wind

In the presence of heavy winds that cause the camera to shake, the object detection algorithm can fail. Indeed, if the camera shakes, the (grayscale) pixels in each frame can be slightly different in brightness from those of the previous, leading to a non-zero difference frame (see section 3.4.1) even where there is no actual motion. In most cases, the thresholding operation (see section 3.4.3) will have a white threshold high enough to not cast these regions to white in the final difference frame, and they will not affect contour detection. However, if the shaking of the camera is substantial enough, these regions of false motion will be reflected in the final difference frame and their contours will be detected.

Generally, when this occurs, a very large contour taking up most of the frame will be detected for a brief period. Since the entrance and exit zones are near the edges of the frame, it is unlikely that the contour's centroid will be located in one of the zones, so it should not cause any false counts on its own. However, if there are currently any actual object contours in the frame, it will cause them to disappear (by merging with the false contour), then reappear. This will at least break contour tracking (i.e. the proper origin and exit point of the particular contour will not be written to the log file). If the object contours are currently located in entrance/exit zones, it could cause counting issues, such as double counts or missed counts. **Figure 54** below shows the potential consequences when false contours are detected in excessive winds:

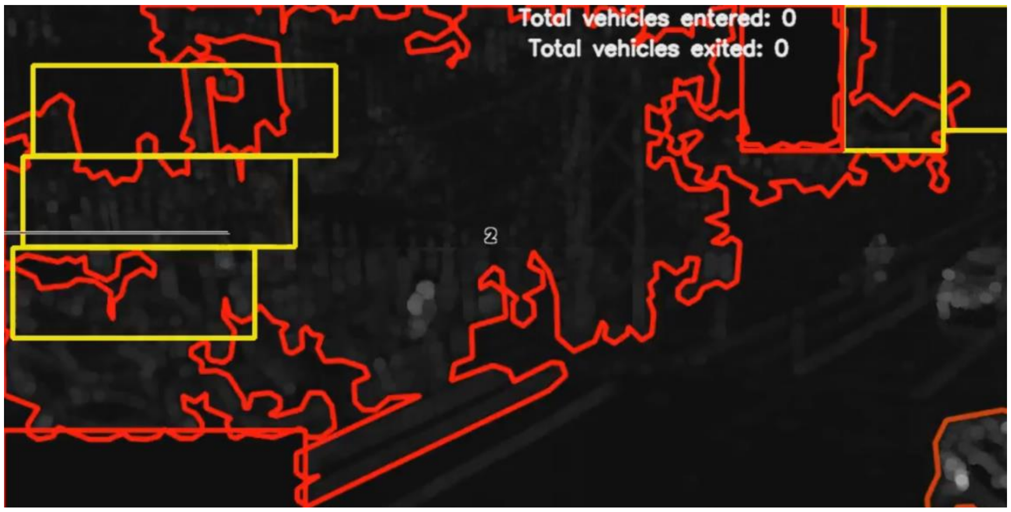


Figure 54: False contours being detected due to heavy winds obscuring designated entrance zones

4.2.3 Comparison to other software

To compare against this project's traffic detection software that uses motion detection, the "Vehicle Detection, Tracking and Counting" software written by Andrews Sobral in 2014 (and updated in 2017) was used [34]. It is the closest software to the one developed in this project: both feature vehicle detection using motion detection and vehicle tracking, and count vehicles when they pass through a region of interest. There are, however, a few differences in implementation. The reference software does not feature distinction between pedestrians and vehicles; it seems to detect contours of any moving object. Furthermore, it has a much more rudimentary way of counting objects: instead of monitoring contours that appear and disappear, it simply adds to a count when an object's centroid crosses a region of interest (line). Unfortunately, this approach only works for vehicle detection (because each lane is one-way); for pedestrian detection, it is impossible to tell whether a pedestrian that crossed a line walked in or out of the frame. Therefore, the reference software was only tested using the above vehicle detection clips, not the pedestrian one.

The configuration for testing the reference software is the following: the position of each line was set where it appeared to be most appropriate (which could be suboptimal because the intricate details of the functioning of the reference software are not known). However, two main sources of error were avoided: the lines were far enough from the edges of the frame so that incoming vehicles would be detected well in advance of crossing the line, and lines were placed such that the number of pedestrians crossing them would be minimized. Furthermore, there did not appear to be a way to set multiple lines at once, so different instances of the software were ran concurrently, each with a line that monitored a designated entrance/exit zone (the lines were placed in a way that minimizes vehicles crossing two different lines). For the video clips, the exact same ones as for this project's software were used, but they were scaled down by a factor of 1:2 because the reference software operated far slower than real-time at the original resolution; this did not affect the vehicle count. The testing results are the following (Table 5):

Table 5: Motion detection results (using reference software [34]):

Stream name	Stream info	Obj. enter (cnt.)	Obj. enter (act.)	Obj. exit (cnt.)	Obj. exit (act.)	Consistency	Accuracy (enter)	Accuracy (exit)
Example 1 (vehicles)	172.168.10.32 14 June 2017 08:55-09:00	203	52	99	50	32%	-190.38%	2%

Example 2 (vehicles)	172.168.10.32 14 June 2017 12:10-12:15	112	50	74	52	56.52%	-24%	57.69%
Example 3 (vehicles)	172.168.10.32 14 June 2017 12:30-12:35	140	60	93	56	62.45%	-33.33%	33.93%
Example 5	Clip included with reference software [34], duration 00:12	5	5	5	5	100%	100%	100%

The performance of the reference software in this project's environment is very poor (in some cases, it achieves a negative accuracy, which means that the percentage error between the count and actual values is higher than 100%!) In comparison, this project's software has no accuracy result below 90% (percentage error smaller than 10%). Indeed, the reference software seems to be optimized only for cases such as the included test file, where there is a clear top-down view of each lane, only vehicles are present, and they are constantly moving in the same direction at moderate speeds. It fails completely when faced with an environment where there are pedestrians, cyclists, and vehicles present, and where vehicles often move very slowly or stop and then continue moving. These factors explain the software's poor performance: it tends to count far more vehicles than there actually are, partly due to counting pedestrians and cyclists as vehicles, and partly because vehicles that cross the detection region of interest (line) very slowly are counted multiple times. This project's software is therefore clearly superior to other available software: it performs far better in the actual environment, while easily matching the performance of the reference software in its own optimal test case (it achieved a consistency and accuracy of 100% in Example 5 after only a few tweaks).

4.3 Software performance and detection accuracy with different resolutions and frame rates

4.3.1 Resolution and frame rate sensitivity

It would be very practical if lower quality video files could be used for motion detection purposes, since continuously streaming video at the source quality (1280x720, 25 FPS) takes a significant amount of bandwidth, which, multiplied by the number of intersections monitored, can be excessive. However, since the integrity of the software accuracy is the most important metric, the parking and traffic count programs must be tested with different resolutions and frame rates, and the recommendation to reduce footage quality cannot be made if any significant degradation in accuracy is found.

For traffic detection, the test files were the same as the ones used to obtain the initial results (see section 4.2). Since vehicles are much faster than pedestrians and should be more impacted by any change in footage, only the vehicle test cases (Examples 1, 2, and 3) were tested. The footage (originally 1280x720, 25 FPS) was re-encoded at lower resolutions and frame rates using the FFmpeg software, in the following manner:

- Resolution scaling: converted video file to target resolution to lower image quality, then converted back to 1280x720 to keep software parameters consistent
- Frame rate scaling: converted video file to target frame rate without increasing video speed (by dropping rest of frames)

The results of the software accuracy tests at different resolutions and frame rates are shown in the following tables (Table 6 and Table 7*)*, as well as the following graphs and analyzed below:

Table 6: Comparison of accuracy of motion detection software at different resolutions

Example 1 (52 entered, 50 exited)	Original (720p)	Scaled 80%	Scaled 60%	Scaled 40%	Scaled 20%
Entered	60	59	53	54	49

Exited	47	48	49	53	55
% error entered	15.38	13.46	1.92	3.85	5.77
% error exited	6.00	4.00	2.00	6.00	10.00
Bit rate (kbps)	6131	1866	1472	1088	771
Example 2 (51 entered, 51 exited)	Original (720p)	Scaled 80%	Scaled 60%	Scaled 40%	Scaled 20%
Entered	51	51	50	46	45
Exited	47	52	55	53	58
% error entered	0.00	0.00	1.96	9.80	11.76
% error exited	7.84	1.96	7.84	3.92	13.73
Bit rate (kbps)	6133	1432	1048	740	515
Example 3 (60 entered, 56 exited)	Original (720p)	Scaled 80%	Scaled 60%	Scaled 40%	Scaled 20%
Entered	49	53	50	53	40
Exited	56	55	58	59	61
% error entered	18.33	11.67	16.67	11.67	33.33
% error exited	0.00	1.79	3.57	5.36	8.93
Bit rate (kbps)	6132	1556	1182	862	627

Note: These measurements are made with a different version of the software, so results may differ from those presented in section 4.2.1

Although it is not immediately obvious which resolution provides the best result, a trend can be detected. When reducing the resolution, the best overall accuracy (considering entered and exited counts) occurs at the 80% scaling mark (see **Figure 56** and **Figure 57**), although Example 1 (see **Figure 55**) performs significantly better at the 60% mark. This is likely due to insufficient performance of the test machine and not a flaw in the algorithm; for a possible explanation of this phenomenon, refer to software performance analysis (section 4.3.2.3). However, accuracy is generally good at around 60% or 80% of original resolution (depending on clip), with a percentage error generally under 10%, so resolution can be decreased up to that level without affecting accuracy significantly. Transcoding the footage at 80 or 60% of the original resolution also reduces the bit rate from 6132 kbps (average) to 1618 kbps or 1234 kbps, or a 73.6% to 79.9% reduction in file size.

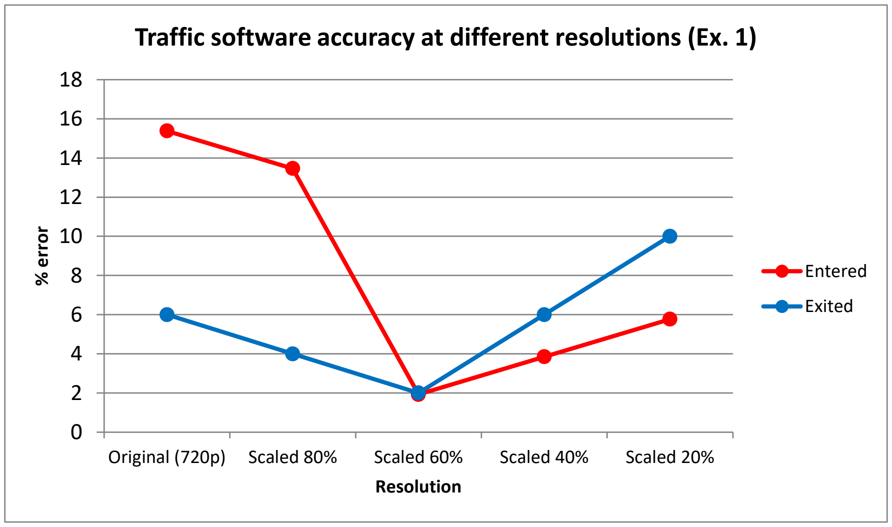


Figure 55: Traffic software accuracy at different resolutions (Example 1)

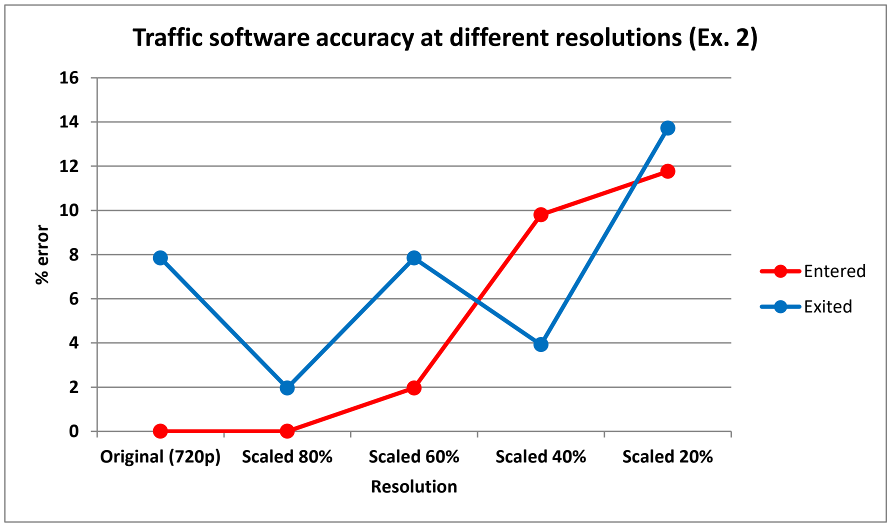


Figure 56: Traffic software accuracy at different resolutions (Example 2)

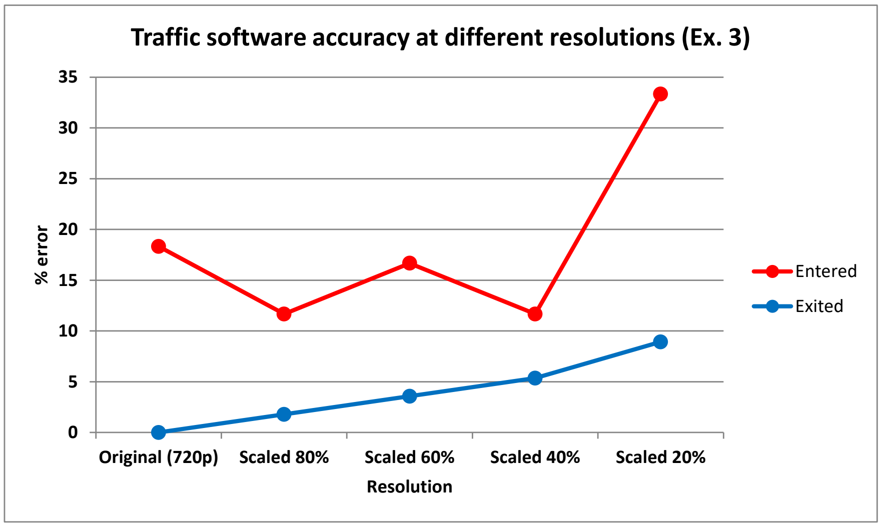


Figure 57: Traffic software accuracy at different resolutions (Example 3)

Table 7: Comparison of accuracy of motion detection software at different frame rates

Example 1 (52 entered, 50 exited)	Original (25 FPS)	80% (20 FPS)	60% (15 FPS)	40% (10 FPS)	20% (5 FPS)
Entered	60	57	50	48	29
Exited	47	47	47	49	37
% error entered	15.38	9.62	3.85	7.69	44.23
% error exited	6.00	6.00	6.00	2.00	26.00
Bit rate (kbps)	6131	3576	3321	2965	2528
Example 2 (51 entered, 51 exited)	Original (25 FPS)	80% (20 FPS)	60% (15 FPS)	40% (10 FPS)	20% (5 FPS)
Entered	51	51	50	48	24
Exited	47	50	51	45	36

% error entered	0.00	0.00	1.96	5.88	52.94
% error exited	7.84	1.96	0.00	11.76	29.41
Bit rate (kbps)	6133	3165	2977	2539	2212
Example 3 (60 entered, 56 exited)	Original (25 FPS)	80% (20 FPS)	60% (15 FPS)	40% (10 FPS)	20% (5 FPS)
Entered	49	49	49	48	33
Exited	56	54	53	51	34
% error entered	18.33	18.33	18.33	20.00	45.00
% error exited	0.00	3.57	5.36	8.93	39.29
Bit rate (kbps)	6132	3251	3037	2661	2320

Note: These measurements are made with a different version of the software, so results may differ from those presented in section 4.2.1

Reducing the frame rate has a much less consistent effect on detection accuracy results than reducing resolution. The sweet spot for each clip is different: for Example 1, the lowest overall error is at 40% or 10 FPS (see Figure 58), for Example 2 it is between 60% to 80% or 15 to 25 FPS (see Figure 59), and for Example 3 it is at 100% or 25 FPS (see Figure 60). The only conclusion is that at 5 FPS does the accuracy decrease significantly, with percentage errors jumping up to 50% at that level. Furthermore, reducing frame rates does not have quite the same effect as reducing resolution: even at 20% FPS, the average bit rate only goes from 6132 kbps to 2353.3 kbps, or only a 61.6% reduction of file size.

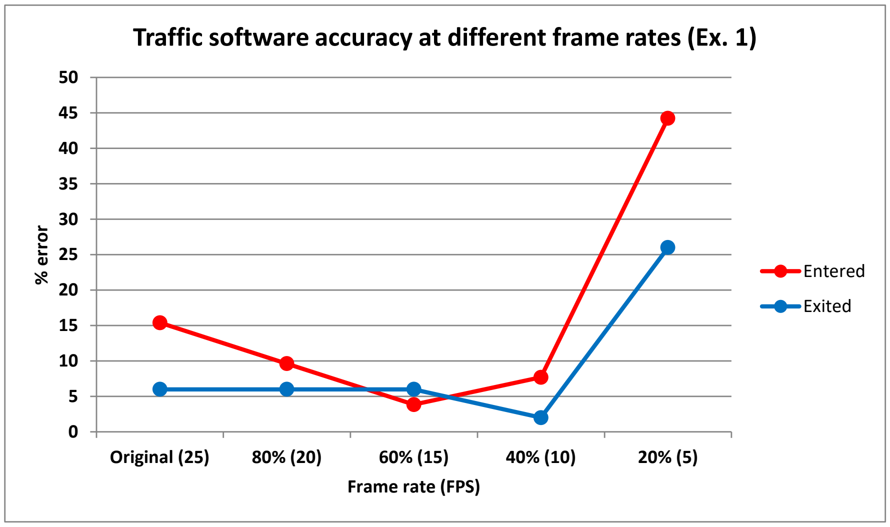


Figure 58: Traffic software accuracy at different frame rates (Example 1)

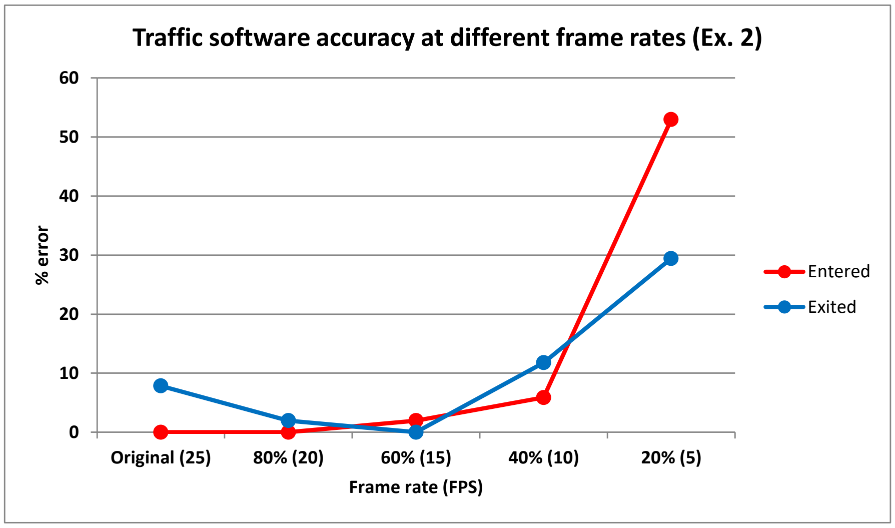


Figure 59: Traffic software accuracy at different frame rates (Example 2)

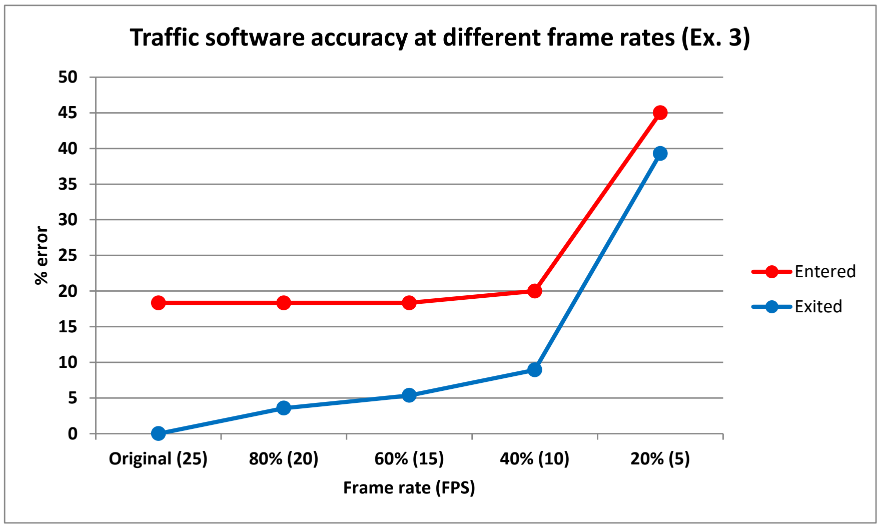


Figure 60: Traffic software accuracy at different frame rates (Example 3)

Considering the above results and the effects on bit rate, the best suggestion for reducing file size in the traffic detection application is scaling the video down to 60% to 80% frame rate (a more accurate percentage may be obtained by testing more resolution scaling factors with more clips). In addition to improving the detection accuracy, lowering the recording quality via resolution decreases bit rates by approximately 80%, and could increase the amount of footage that can be transferred over the same network connection by a factor of approximately 5. However, for the sake of reducing the file size, the frame rate should not be altered: the reduction in bit rate is much less than by changing the resolution and there does not seem to be a setting where the majority of clips perform best. It would be possible to reduce both the resolution and frame rate, but once again this is not recommended due to the unpredictable effect of changing the frame rate and the relatively low impact on video file size.

As for parking detection, the test files were chosen from the video clips used for accuracy testing (see section 4.1). Among those, the clips where the software developed in this project made detection mistakes (Clip 4, Clip 8, Clip 14) were chosen because using them will reveal potential accuracy improvements from reducing resolution or frame rate. The footage (originally 1280x720, 10 FPS) was re-encoded at lower resolutions and frame rates using the FFmpeg software in the same manner as for the motion detection tests. The results of the software accuracy tests at different resolutions and frame rates are shown in Table 8, Table 9 and graphs (Figure 61 *and* Figure 62) and analyzed below:

Table 8: Comparison of accuracy of parking detection software at different resolutions

Clip 4	Original (720p)	Scaled 80%	Scaled 60%	Scaled 40%	Scaled 20%
Mistakes	1	1	1	2	3
% Mistakes	9.09	9.09	9.09	18.18	27.27
Bit rate (kbps)	1998	880	746	544	339
Clip 8	Original (720p)	Scaled 80%	Scaled 60%	Scaled 40%	Scaled 20%
Mistakes	1	0	0	1	5
% Mistakes	9.09	0	0	9.09	45.45
Bit rate (kbps)	2169	806	657	477	311
Clip 14	Original (720p)	Scaled 80%	Scaled 60%	Scaled 40%	Scaled 20%
Mistakes	1	1	1	3	2
% Mistakes	9.09	9.09	9.09	27.27	18.18
Bit rate (kbps)	1576	706	561	381	235

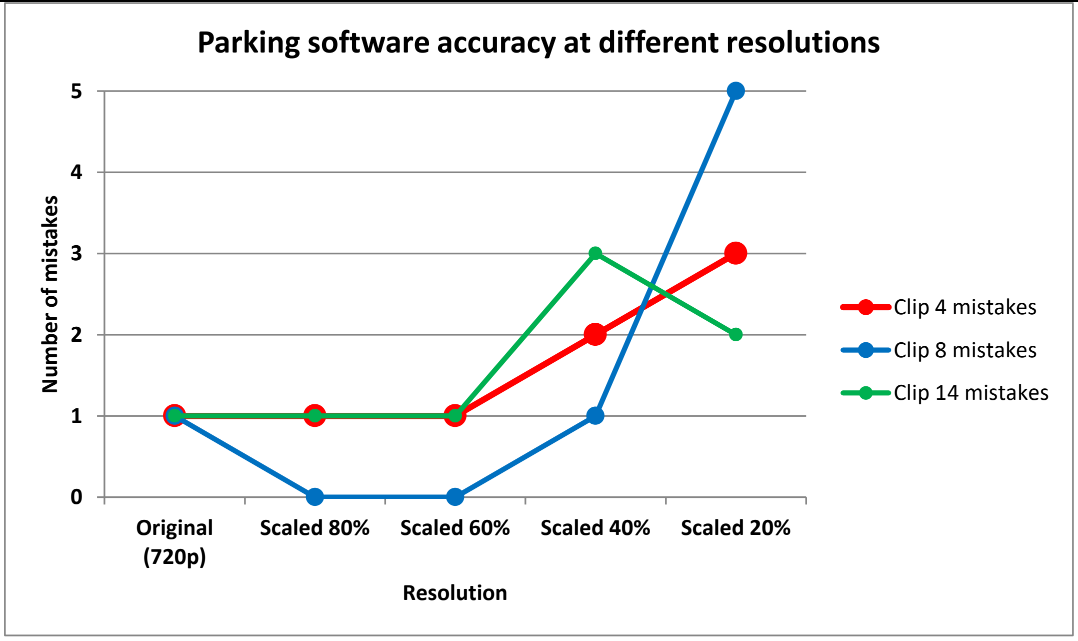


Figure 61: Parking software accuracy at different resolutions

In terms of reducing resolution, it seems that scaling all the way down to 80% or even 60% does not affect results negatively, and there is even a small improvement in Clip 8, with the number of mistakes decreasing to 0 (see Figure 61). However, when observing the detection results in the output frame, it is clear that decreasing the resolution reduces the accuracy: at 100% resolution, most parking spot occupancy is detected with 4 out of 4 polygons being the correct state (see section 3.3.1), while at 80% and 60%, the occupancy is often detected with 3 or even only 2 polygons being the correct state, so there is a much higher risk of eventual detection error. Furthermore, in Clip 8, the detection mistake is due to non-ideal camera setup (text overlay overlapping the parking spot and creating artificial edges, see section 5.1.1), so it is not a true detection mistake, and the fact that it does not occur at lower resolutions simply shows that the Laplacian algorithm is becoming less reliable as resolution decreases. In summary, even though numerical results indicate that reducing resolution to 60% will reduce bit rate significantly (from an average of 1914.3 kbps down to 654.7 kbps at 60%, or a 65.8% improvement) while not decreasing accuracy, the recommendation to reduce resolution cannot be made when considering actual observations.

Table 9: Comparison of accuracy of parking detection software at different frame rates

Clip 4	Original (10)	80% (8)	60% (6)	40% (4)	20% (2)
Mistakes	1	1	1	1	1
% Mistakes	9.09	9.09	9.09	9.09	9.09
Bit rate (kbps)	1998	1221	1106	1012	898
Clip 8	Original(10 FPS)	80% (8 FPS)	60% (6 FPS)	40% (4 FPS)	20% (2 FPS)
Mistakes	1	1	1	1	1
% Mistakes	9.09	9.09	9.09	9.09	9.09
Bit rate (kbps)	2169	1232	1170	1074	957
Clip 14	Original (10)	80% (8)	60% (6)	40% (4)	20% (2)
Mistakes	1	1	0	0	0
% Mistakes	9.09	9.09	0.00	0.00	0.00
Bit rate (kbps)	1576	995	960	885	859

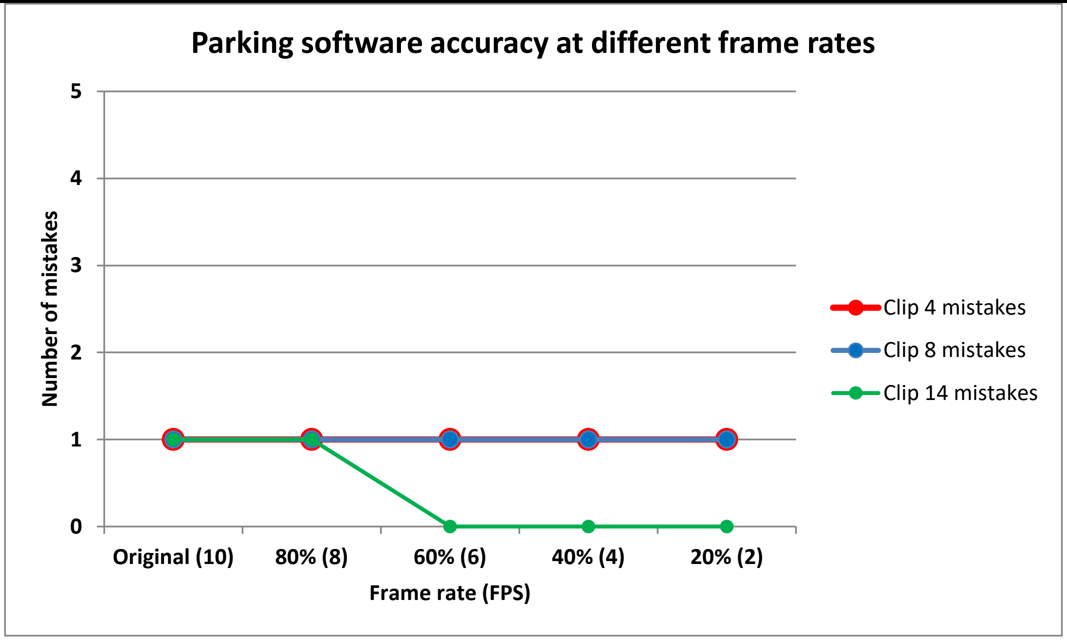


Figure 62: Parking software accuracy at different frame rates

As for reducing the frame rate, the results are very consistent, with almost every clip remaining at 1 detection mistake. Although Clip 14 shows an improvement at 6 FPS and below, with the single detection error disappearing (see Figure 62), this is purely due to a very improbable timing anomaly: the sampling of parking spot detection (see section 3.3.1) occurred at slightly different frames and the parking spot remained at 2/4 occupancy for the rest of the test, so there was no chance to recover from the error. In reality, the detection results in terms of parking polygon status are not affected. More than approximately 2 frames per second of data is simply wasted for the parking detection software, so reducing the frame rate all the way down to 2 can reduce bit rates from an average of 1914.3 kbps to 904.7 kbps, or a 52.74% reduction, without affecting results in the great majority of scenarios.

4.3.2 Software performance and hardware required

Significant savings could be obtained if the parking and motion detection programs could run on low-cost hardware, especially if the solution ends up being deployed en masse. Even better would be the possibility of performing the parking detection and traffic count on-site with a low-power IoT device (such as a Raspberry Pi), which would save tremendous amounts of bandwidth, as the video footage will not have to be transmitted over the network. However, the recommendation to use low-end hardware cannot be made if it can lead to worse results than what the results demonstrate, so the performance of the software must be evaluated on the current hardware by monitoring the following key metrics: frame rate/frame times, and CPU/memory usage.

Performance testing consists of the same test clips as the ones for accuracy (see section 4.3.1). The frame times are recorded for the entire duration and are used to calculate an average frame rate, the processing time in minutes, and the percentage of real-time speed the software achieves. For resource utilization testing, one clip for both parking and motion detection is run at all resolution and frame rate configurations, and software CPU and RAM usage are recorded every 10 seconds. The test system is a VMWare virtual machine that has been allocated 4 cores of an Intel® Xeon(R) CPU E5-2650 v2 @ 2.60GHz, has 15.5 GiB of RAM, and runs Linux CentOS 7 64-bit. The performance results are shown in tables and the average results for all clips are displayed in charts in section 4.3.2.1. The resource utilization results are displayed in tables and graphs in section 4.3.2.2, then analyzed.

4.3.2.1 Software performance

Table 10: Performance of traffic software at different resolutions

Example 1	Original (720p)	Scaled 80%	Scaled 60%	Scaled 40%	Scaled 20%
Average FPS	8.325	8.335	8.337	8.336	8.337
Processing time	5.005	5.005	5.004	5.004	5.004
% of real-time speed	99.90	99.90	99.92	99.92	99.92
Example 2	Original (720p)	Scaled 80%	Scaled 60%	Scaled 40%	Scaled 20%
Average FPS	8.325	8.336	8.337	8.337	8.337
Processing time	5.005	5.002	5.002	5.002	5.002
% of real-time speed	99.90	99.96	99.97	99.97	99.97
Example 3	Original (720p)	Scaled 80%	Scaled 60%	Scaled 40%	Scaled 20%
Average FPS	8.327	8.364	8.365	8.365	8.365
Processing time	5.004	5.002	5.001	5.001	5.001
% of real-time speed	99.93	99.97	99.98	99.99	99.98

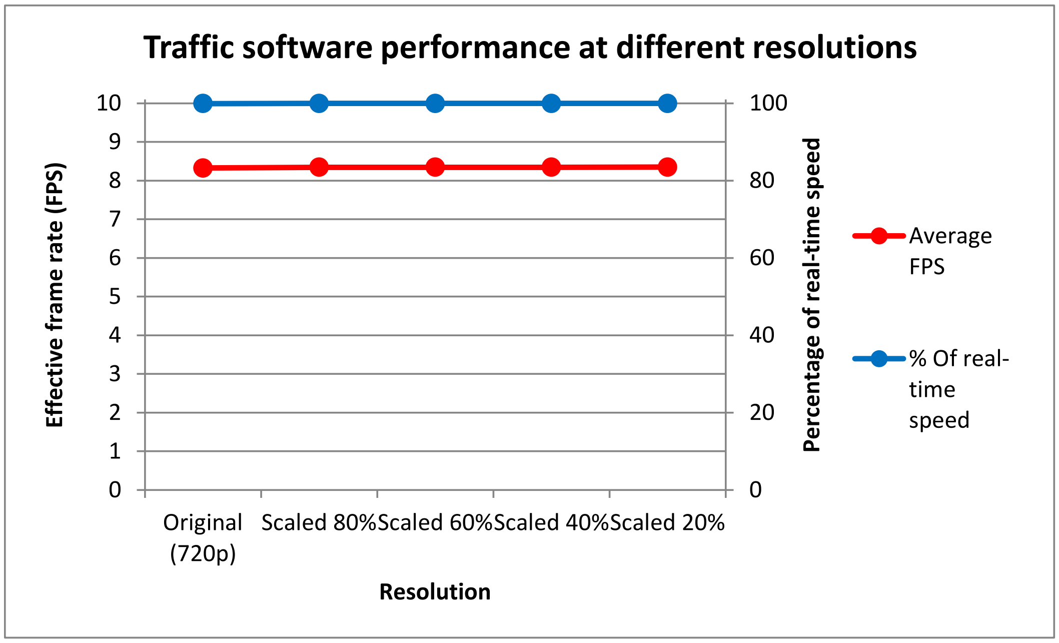


Figure 63: Performance of traffic software at different resolutions

From the data in Table 10 and Figure 63, the performance of the traffic software does not improve when resolution is decreased, since the software already operates essentially in real-time, at an effective frame rate of nearly 8.33 FPS (which corresponds to the 25 FPS of the original clip, considering that the motion detection algorithm uses only one-third of the frames). Therefore, the recommended method for reducing file size (see section 4.3.1) does not simultaneously improve software performance, but only because performance is already at its maximum.

Table 11: Performance of traffic software at different frame rates

Example 1	Original (25 FPS)	80% (20 FPS)	60% (15 FPS)	40% (10 FPS)	20% (5 FPS)
Average FPS	8.325	6.668	5.000	3.333	1.667
Processing time	5.005	5.004	5.006	5.005	5.009
% of real-time speed	99.90	99.91	99.87	99.90	99.82
Example 2	Original (25)	80% (20)	60% (15)	40% (10)	20% (5)
Average FPS	8.325	6.668	5.000	3.334	1.667
Processing time	5.005	5.004	5.006	5.005	5.009
% of real-time speed	99.90	99.91	99.87	99.91	99.83
Example 3	Original (25)	80% (20)	60% (15)	40% (10)	20% (5)
Average FPS	8.327	6.668	5.000	3.333	1.667
Processing time	5.004	5.004	5.006	5.005	5.009
% of real-time speed	99.93	99.91	99.87	99.90	99.82

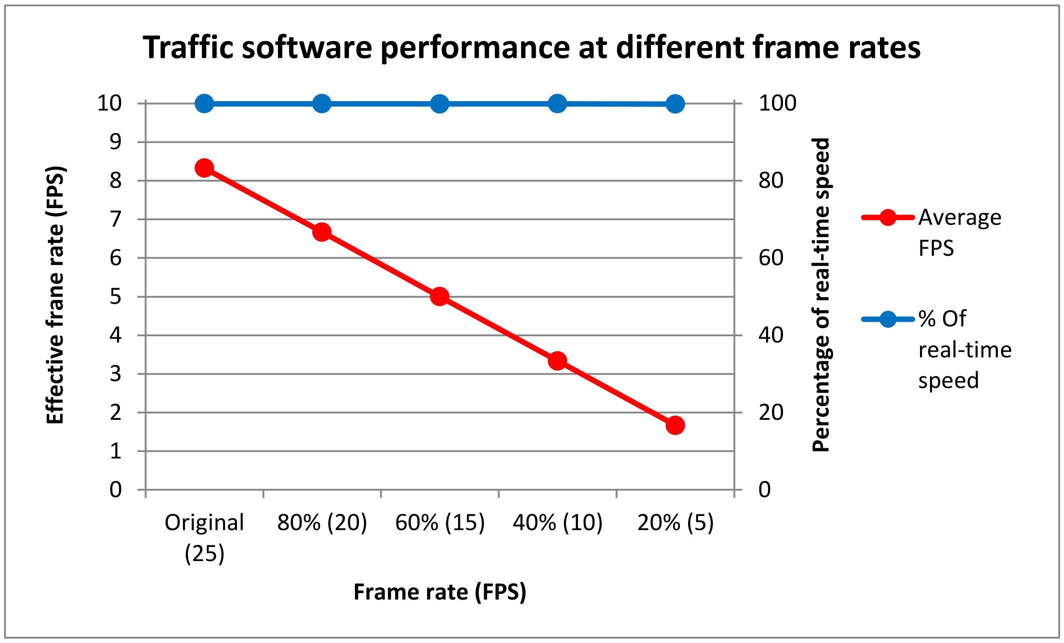


Figure 64: Performance of traffic software at different frame rates

From the data in Table 11 and Figure 64, the performance of the traffic software remains consistently at real-time speed, with an effective frame rate that is always one-third of the original clip FPS (8.33 FPS effective at 25 FPS, 6.67 FPS effective at 20 FPS, 5 FPS effective at 15 FPS, 3.33 FPS effective at 10 FPS, 1.67 FPS effective at 5 FPS).

Even though real-time operation has been achieved on the test machine, the recommendation to run the software on a much slower Raspberry Pi or similar machine cannot be made in good faith. Indeed, even if only a small percentage of the frames were skipped, some vehicle appearances or disappearances in an entrance or exit gate will be missed by the software, since these events sometimes last only a very brief amount of time. This can be inferred from the lower detection accuracy when reducing frame rates (see section 4.3.1).

Table 12: Performance of parking software at different resolutions

Clip 4	Original (720p)	Scaled 80%	Scaled 60%	Scaled 40%	Scaled 20%
Average FPS	4.009	6.391	8.872	9.997	10.012
Processing time	1.663	1.043	0.751	0.667	0.666
% of real-time speed	40.09	63.91	88.72	99.97	100.12
Clip 8	Original (720p)	Scaled 80%	Scaled 60%	Scaled 40%	Scaled 20%
Average FPS	3.769	6.493	8.895	10.005	10.008
Processing time	3.131	2.125	1.387	1.233	1.232
% of real-time speed	37.69	58.03	88.95	100.05	100.08
Clip 14	Original (720p)	Scaled 80%	Scaled 60%	Scaled 40%	Scaled 20%
Average FPS	3.854	6.354	8.796	10.000	10.014
Processing time	1.297	0.787	0.568	0.500	0.499
% of real-time speed	38.54	63.54	87.96	100.00	100.14

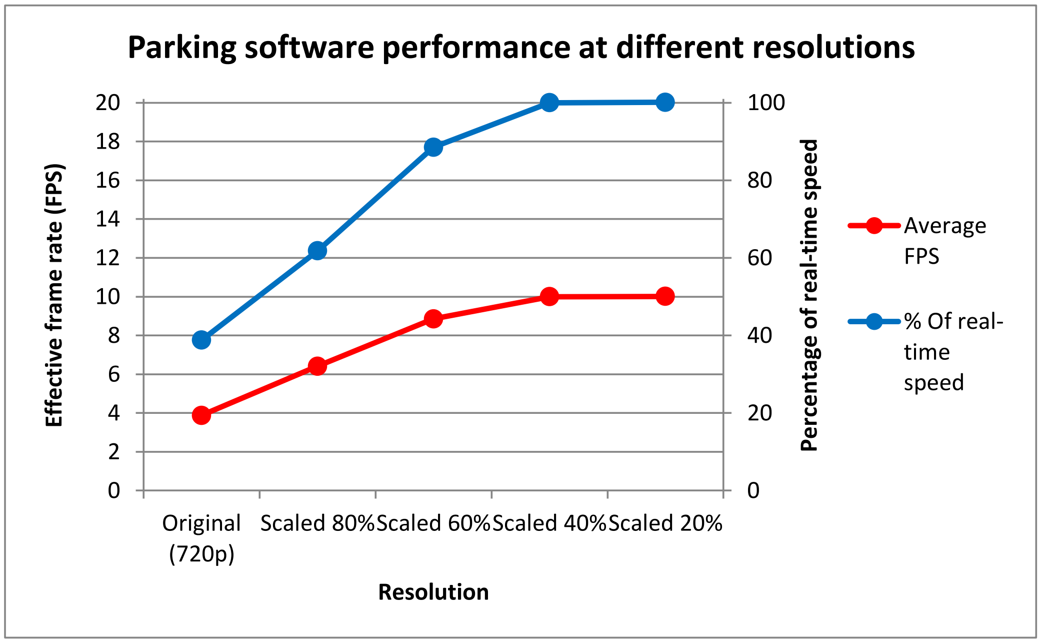


Figure 65: Performance of parking software at different resolutions

Table 13: Performance of parking software at different frame rates

Clip 4	Original (25)	80% (20)	60% (15)	40% (10)	20% (5)
Average FPS	4.009	4.002	4.004	3.993	2.009
Processing time	1.663	1.333	0.999	0.668	0.664
% of real-time speed	40.09	50.02	66.73	99.84	100.44
Clip 8	Original (25 FPS)	80% (20 FPS)	60% (15 FPS)	40% (10 FPS)	20% (5 FPS)
Average FPS	3.769	3.841	3.836	3.838	2.004
Processing time	3.131	2.569	1.929	1.286	1.231
% of real-time speed	37.69	48.01	63.93	95.94	100.22
Clip 14	Original (25)	80% (20)	60% (15)	40% (10)	20% (5)
Average FPS	3.854	3.834	3.860	3.842	2.011
Processing time	1.297	1.043	0.777	0.521	0.497
% of real-time speed	38.54	47.93	64.33	96.05	100.56

From the results in Table 12 and Figure 65, reducing video resolution linearly increases parking detection software performance in a linear fashion (4 FPS or 40% of real-time at full resolution, 6.4 FPS or 64% of real-time at 80% resolution, 8.8 FPS or 88% of real-time at 60% resolution), until it reaches 10 FPS, the frame rate of the video clip (at 40% of original resolution and below), and tapers there because the software operates at 100% of real-time speed.

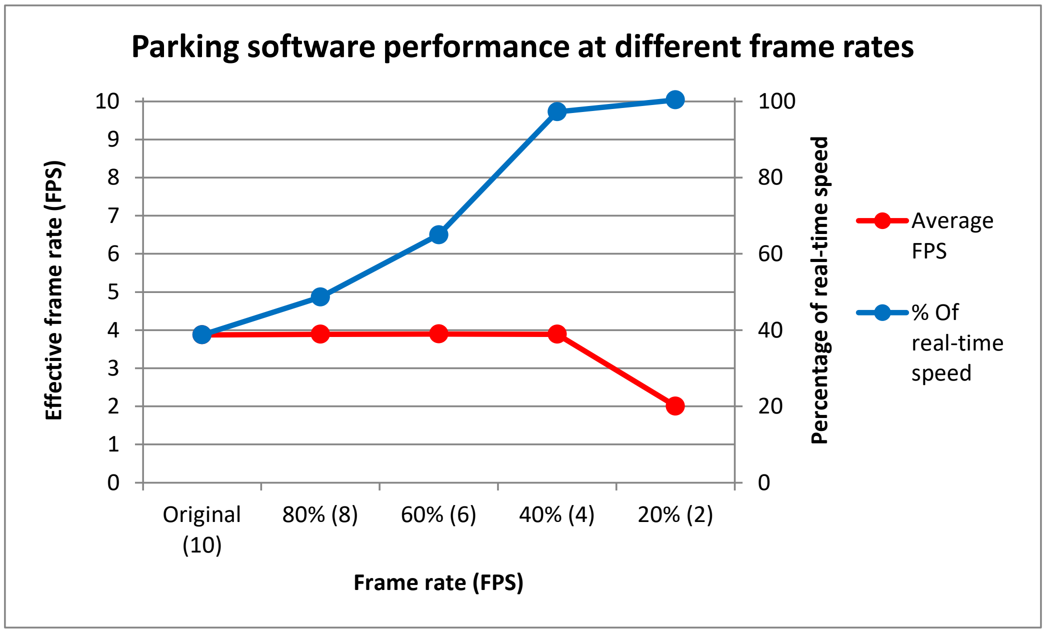


Figure 66: Performance of parking software at different frame rates

From the results in Table 13 and Figure 66, reducing the frame rate does not change the frame rate of the parking software, which stays a hair under 4 FPS, until the frame rate falls below that value and forces the software to run at that speed. However, since the frame rate decreases, the percentage of real-time performance the software can achieve increases, until it reaches 100% at approximately 4 FPS and remains at that level. Therefore, reducing frame rate, the recommended way to decrease video file size for parking detection (see section 4.3.1) also brings the software performance closer to real-time, so it is highly recommended.

The parking detection frame rate numbers seem very low, especially at high resolutions, but are adequate for this type of application because there are no very brief events: it generally does not matter if the software misses a split-second event. Although this is still acceptable on the current system, it shows that a low-power IoT system similar to a Rasbperry Pi could definitely not run the parking software at anywhere near real-time speeds and the accuracy of its results will likely suffer as a result, so running the program on such a device cannot be recommended.

4.3.2.2 Hardware utilization

Table 14: CPU and RAM utilization by traffic detection software at different resolutions (Example 1)

Example 1	Original (720p)	Scaled 80%	Scaled 60%	Scaled 40%	Scaled 20%
CPU average (%)	31.75	19.5	10.5	5.05	1.65
CPU maximum (%)	38	27	16	7	2
CPU minimum (%)	27	16	9	4	1
RAM average (MB)	166.73	111.95	70.91	38.45	19.7
RAM maximum (MB)	170.6	119.1	71.3	38.7	19.7
RAM minimum (MB)	161.1	60.3	68	37	19.7

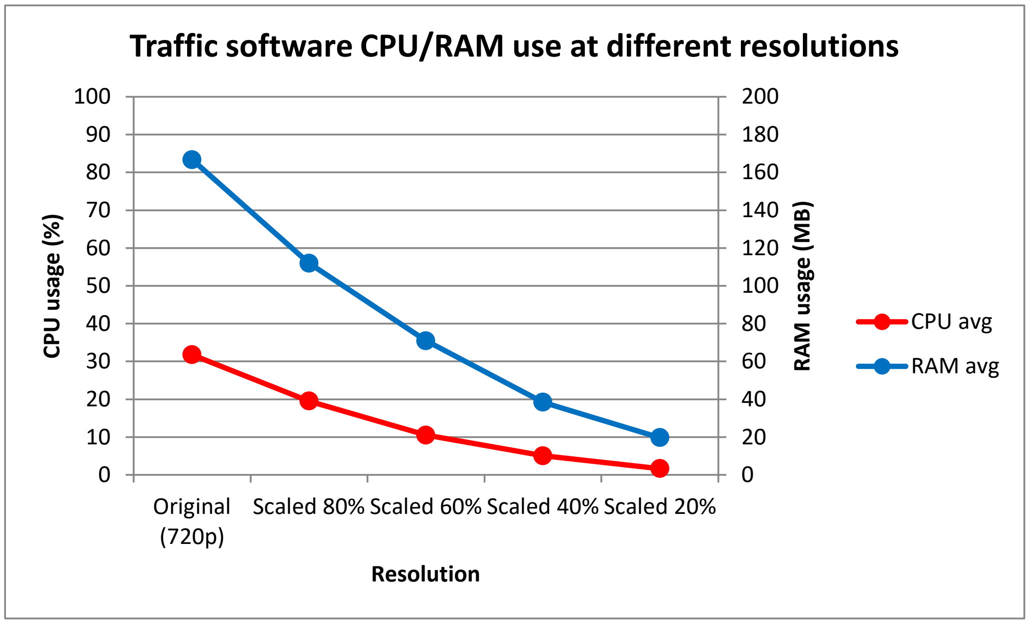


Figure 67: Traffic software CPU/RAM utilization at different resolutions

Table 15: CPU and RAM utilization by traffic detection software at different frame rates (Example 1)

Example 1	Original (25 FPS)	80% (20 FPS)	60% (15 FPS)	40% (10 FPS)	20% (5 FPS)
CPU average (%)	31.75	26.7	20.15	16.65	8.6
CPU maximum (%)	38	34	26	18	9
CPU minimum (%)	27	20	15	15	8
RAM average (MB)	166.73	169.90	173.44	171.76	172.65
RAM maximum (MB)	170.6	173.3	176.2	175.5	175.4
RAM minimum (MB)	161.1	166.2	167.5	166.4	166.4

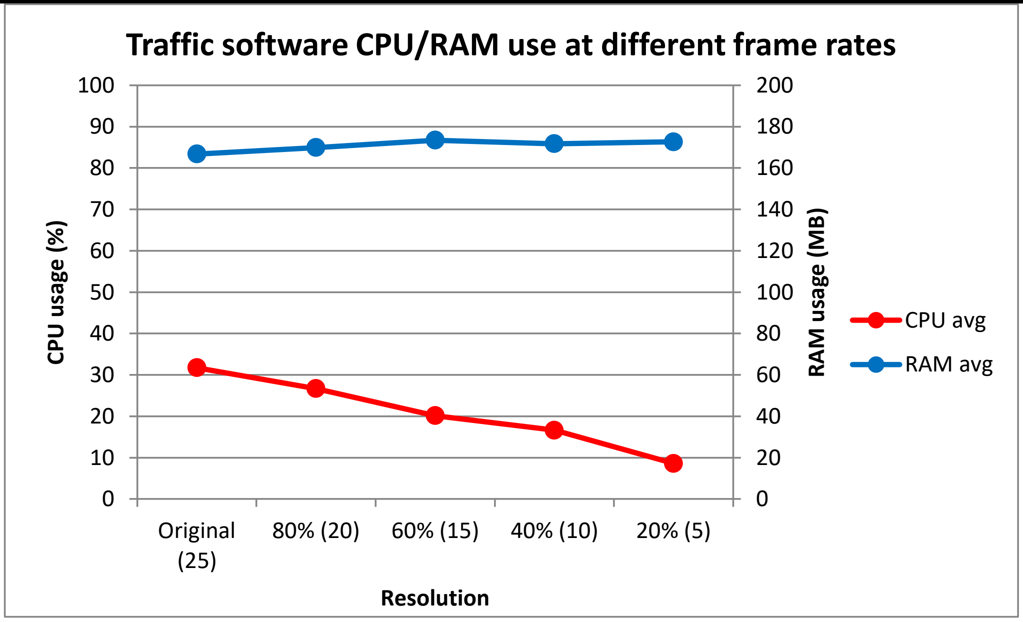


Figure 68: Traffic software CPU/RAM utilization at different frame rates

According to the results in Table 14 and Figure 67, the CPU and RAM utilization decrease in an exponential fashion when resolution decreases: the most dramatic changes occur from 100% of original resolution to 80%. This trend is consistent with the fact that the traffic software performance cannot improve because it already operates in real-time (see section 4.3.2.1). CPU and RAM usage start at approximately 30% and 167 MB (original resolution) and decrease all the way down to 1.65% and 20 MB (20% of original resolution), but accuracy results at this level are very poor (see section 4.3.1). At resolutions ranging from 60% to 80% of the original, where accuracy results are still good (see section 4.3.1), the CPU usage is between 10% and 20%, and the RAM is between 70 MB and 112 MB. As for varying frame rates, according to results in Table 15 and Figure 68, the CPU usage decreases linearly as resolution decreases (from 30% at 25 FPS all the way down to approximately 9% at 5 FPS), and RAM usage is approximately constant at around 170 MB. Since software accuracy does not change much until frame rate falls below 10 FPS (see section 4.3.1), the CPU utilization could be reduced down to around 17%. Due to the multi-threaded aspect of the software, which results in CPU load shared equally between cores, it is possible to extrapolate the minimum number of cores to run each instance of the software in real-time, which should be 3 Intel Xeon cores at original resolution and only 1 core if resolution or frame rate are reduced. RAM usage is between 165 MB and 175 MB, and neither resolution or frame rate variance causes it to change in a systematic manner when the resolution is decreased. In terms of multithreading, all 4 cores share the load nearly equally. Due to the multi-threaded aspect of the software, it is possible to extrapolate the minimum number of cores to run each instance of the software in real-time, which should be 2 Intel Xeon cores, assuming that the recommendation to reduce video size solely through decreasing resolution (see section 4.3.1) is followed and that the CPU usage is approximately 60%.

Table 16: CPU and RAM utilization by parking detection software at different resolutions (Clip 8)

Clip 8	Original (720p)	Scaled 80%	Scaled 60%	Scaled 40%	Scaled 20%
CPU average (%)	26	25.86	24	16	8.5
CPU maximum (%)	26	26	24	16	9
CPU minimum (%)	26	25	24	16	8
RAM average (MB)	65.75	48.9	32.3	23.05	17.05
RAM maximum (MB)	66.1	49.9	32.5	23.1	17.1
RAM minimum (MB)	62.8	47.5	31.5	22.9	16.9

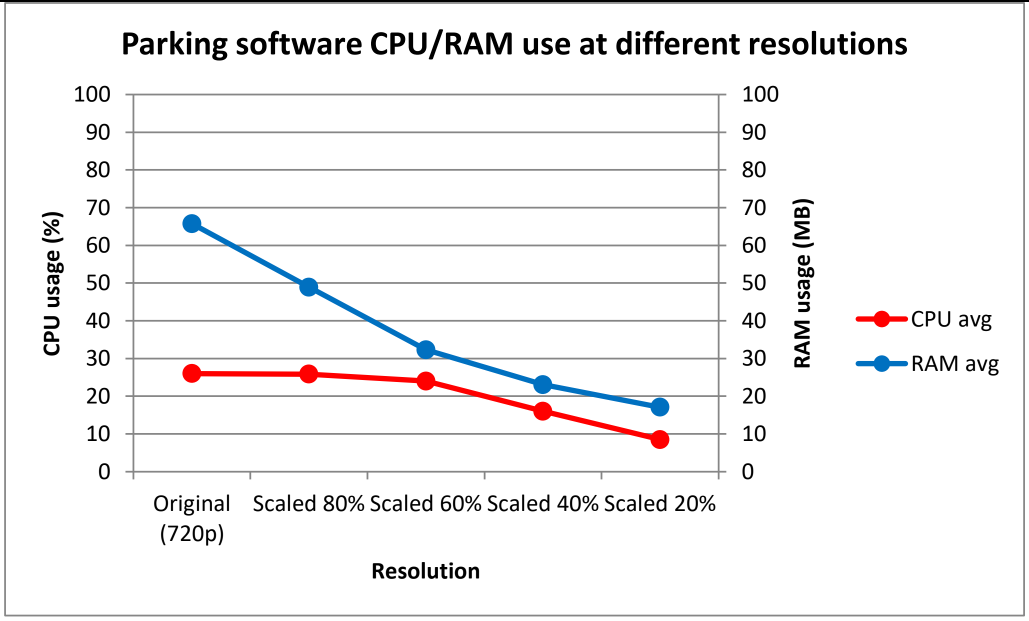


Figure 69: Parking software CPU/RAM utilization at different resolutions

Table 17: CPU and RAM utilization by parking detection software at different frame rates (Clip 8)

Clip 8	Original (25 FPS)	80% (20 FPS)	60% (15 FPS)	40% (10 FPS)	20% (5 FPS)
CPU average (%)	26	26	26.125	26.4	14.75
CPU maximum (%)	26	26	27	27	15
CPU minimum (%)	26	26	26	26	14
RAM average (MB)	65.75	71.01	69.79	68.22	70.28
RAM maximum (MB)	66.1	72.3	70.1	70.3	71.3
RAM minimum (MB)	62.8	68.5	67.6	67.6	69.2

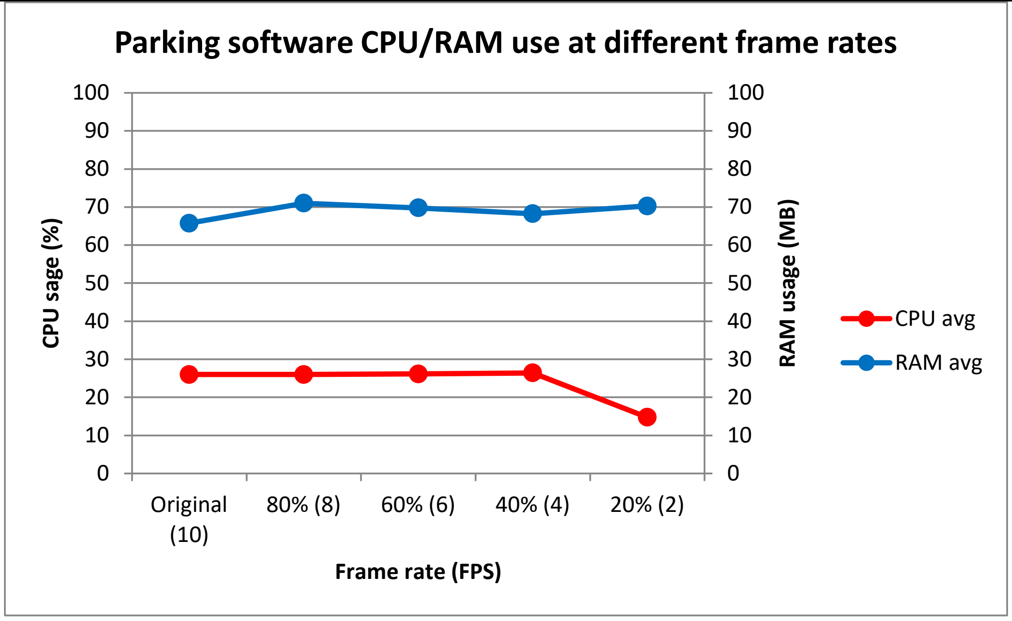


Figure 70: Parking software CPU/RAM utilization at different frame rates

From the data in Table 16 and Figure 69, when changing resolution, the CPU utilization by the parking software was approximately 25% from 100% resolution to 60%, then drops linearly (to 16% at 40% resolution and 8.5% at 20% resolution). This trend makes sense because parking software performance increases linearly until 60% resolution (see section 4.3.2.1), then hits the 10 FPS cap at 40%, which is the point where CPU usage starts to decrease. When varying frame rates, according to data in Table 17 and Figure 70, the CPU usage is very consistent at 26% from 10 FPS to 4 FPS, and drops to 15% at 2 FPS, which is consistent with results in section 4.3.2.1: the software performs at its maximum capacity from 10 FPS to 4 FPS, then is constrained by the video clip frame rate at 2 FPS. The RAM usage was always between 65 and 72 MB, and remained constant throughout testing, the same result as when frame rate was varied in traffic detection. Due to the multi-threaded nature of the software that distributes the load over multiple cores, reducing the number of cores (for example, 1 Xeon core instead of 4) would still likely produce the same performance, especially if the recommendation to reduce the frame rate down to 2 FPS (see section 4.3.1) is followed, since the CPU will be far from being maxed out in that scenario.

4.3.2.3 Motion detection frame time analysis

During motion detection accuracy testing in section 4.3.1, a somewhat implausible result emerged: the accuracy generally improved when resolution was decreased to 80% or even 60%. This should not be the case, since the 100% resolution clip has the most detail, and objects should be clearly visible: results should be better, or at least equal to lower resolution results. However, in Example 1, the percentage error dropped from approximately 15% and 6% to approximately 2% and 2% (entered and exited counts respectively) when resolution decreased from 100% to 60%. Poor software performance due to high resolution could be a cause, but in testing, the average frame rate numbers were nearly identical (see section 4.3.2.1).

However, by analyzing the frame rate over time, focusing on Example 1 (the most egregious case of accuracy increase when reducing resolution), an interesting trend was detected (see **Figure 71** and **Figure 72**). Although the average frame rate was comparable between the 100% and 60% resolution clip, the consistency of frame times, and by extension of frame rate, was much worse in the original resolution clip! These frame rate spikes are the most likely cause of detection inconsistencies, and indicate that the test system is not fast enough to handle the software when running at 100% resolution, even if it appears to run in real-time.

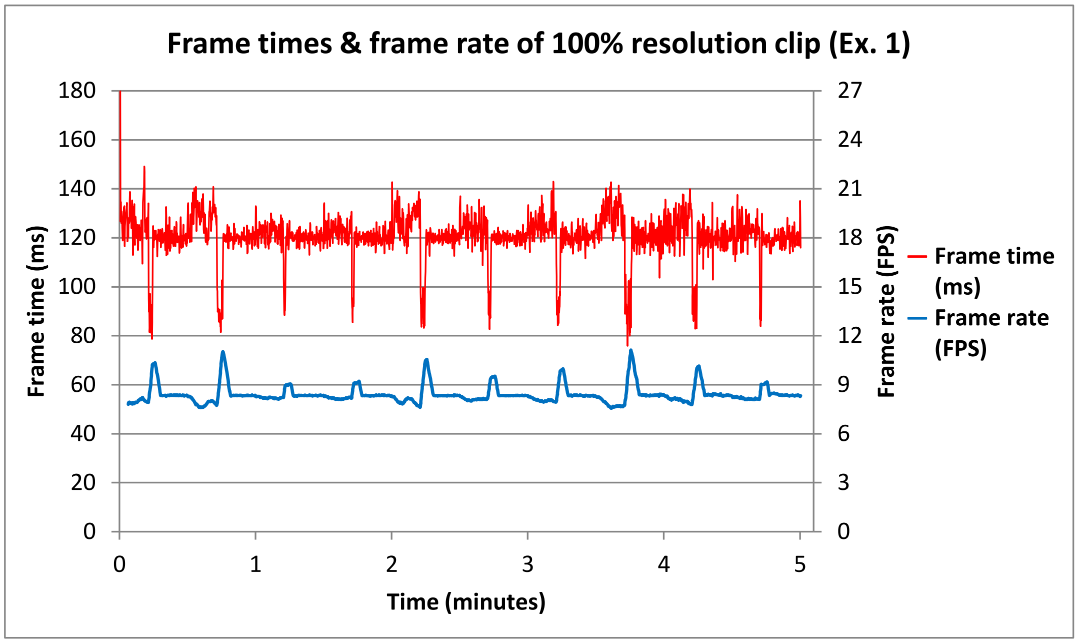


Figure 71: Motion detection frame times and frame rate at 100% resolution (Example 1)

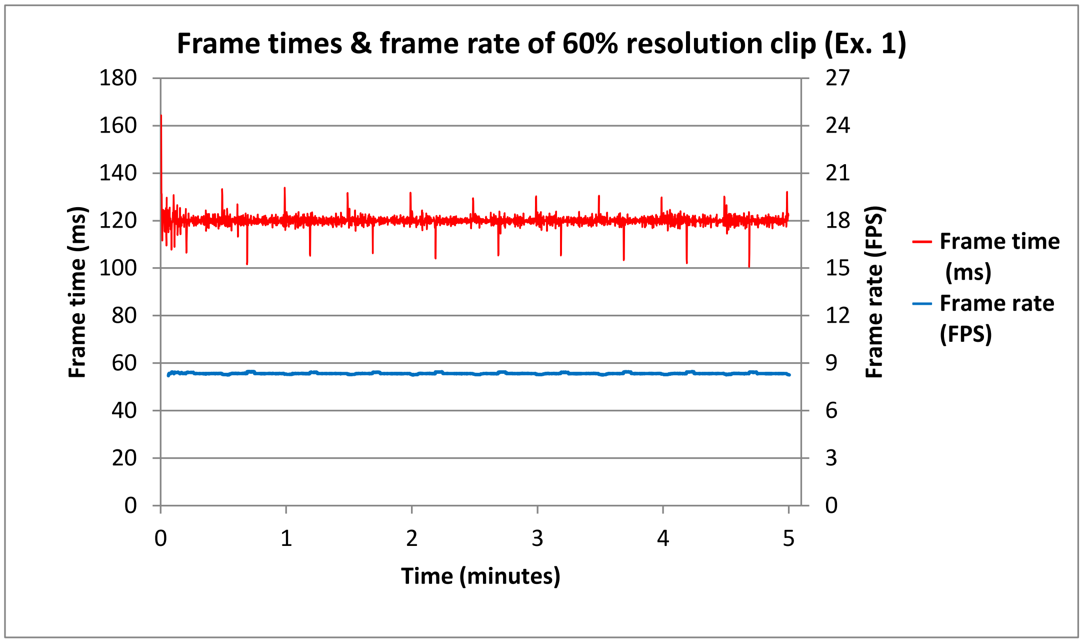


Figure 72: Motion detection frame times and frame rate at 60% resolution (Example 1)

4.3.3 Software performance with double speed footage

To reduce processing times for large amounts of pre-recorded footage, a potential solution would be to speed up the video. However, for motion detection, the viability of this solution in terms of accuracy and performance must be tested before it can be officially recommended. Therefore, the motion detection software will be tested using double speed footage for both consistency and accuracy (metrics defined in section 4.2.1), and hardware performance (processing time, average frame rate, and percentage of frames that can be analyzed in real-time), similarly to tests performed in section 4.3.2. The test clip is a 5-minute, 25 FPS video that will be sped up by a factor of 2. There are two ways of doubling the speed of the video: by dropping half the frames, or by doubling the frame rate. Both will be tested, resulting in a 2.5-minute, 25 FPS clip and a 2.5-minute, 50 FPS one. The results of the accuracy tests are shown in Table 18and the performance results are shown in Table 19, and they are analyzed below:

Table 18: Comparison of accuracy of motion detection software at different video speeds

Example 1 (52 entered, 50 exited)	Entered	Exited	% error enter	% error exit
Original clip	60	47	15.38	6
2x speed (dropped frames)	51	52	1.92	4
2x speed (increased FPS)	63	46	21.15	8
Example 2 (51 entered, 51 exited)	Entered	Exited	% error enter	% error exit
Original clip	51	47	0	7.84
2x speed (dropped frames)	49	46	3.92	9.80
2x speed (increased FPS)	55	53	7.84	3.92
Example 3 (60 entered, 56 exited)	Entered	Exited	% error enter	% error exit
Original clip	49	56	18.33	0
2x speed (dropped frames)	55	51	8.33	8.93
2x speed (increased FPS)	55	58	8.33	3.57

Table 19: Comparison of performance of motion detection software at different video speeds

Example 1	Original clip	2x speed (dropped frames)	2x speed (increased FPS)
Average FPS	8.325	8.323	8.621
Processing time (min)	5.005	2.507	4.837
% of real-time speed	99.90	99.71	51.68
Example 2	Original clip	2x speed (dropped frames)	2x speed (increased FPS)
Average FPS	8.325	8.332	10.674
Processing time (min)	5.005	2.504	3.907
% of real-time speed	99.90	99.83	63.99
Example 3	Original clip	2x speed (dropped frames)	2x speed (increased FPS)
Average FPS	8.327	8.349	10.731
Processing time (min)	5.004	2.507	3.886
% of real-time speed	99.93	99.71	64.34

In Example 2 and Example 3 the lowest error is obtained when doubling the speed by increasing the frame rate instead of skipping frames (in Example 3, the results are even better than the original clip!). In Example 1, the situation is different: doubling the speed by dropping frames provides the best results by far, likely because the software behaves poorly with this particular clip (see section 4.3.2.3 for more details). Overall, the accuracy results favor the speed doubling method with frame rate increase. Therefore, if there is a significant amount of pre-recorded footage that must be analyzed, doubling the speed by increasing the frame rate is an acceptable technique to obtain results that are quite accurate at a faster rate.

However, the speed gains are marginal on the current hardware: even if the clip is half the speed, the software cannot process it in real-time (only between 51.68%-64.34% of real-time speed), and the actual processing time only decreases by a maximum of 22% (from 5 minutes to 3.886 minutes). Indeed, on the current hardware, the motion detection software can only achieve a maximum frame rate of 10.731 FPS, which corresponds to a raw video frame rate of 32.193 FPS. If processing time must truly be reduced to 50% of the original speed, there are two options. Either faster hardware (that can handle 50 FPS video) must be used, or the frame skipping technique can be used (for processing speeds of 2.504-2.507 minutes or almost 100% of real-time speed) at the cost of a less accurate result in general.

4.4 Comparison of OpenCV and MATLAB implementation of traffic detection

4.4.1 Comparison of software architecture and library functions

Both the OpenCV-based traffic detection software (henceforth referred to as the new software) developed in this project and its MATLAB counterpart [40] (henceforth referred to as the reference software) are based on motion detection. They feature a similar overall architecture layout, dividing the problem into three main steps: object detection, tracking, and counting. However, the implementation of each of these steps differs significantly between the two programs.

4.4.1.1 Object detection

For object detection, the new software isolates foreground (moving) objects from the background by performing an absolute difference operation using two or three consecutive frames (see section 3.4.1). The reference software uses a different approach: it detects moving objects using foreground detection (vision.ForegroundDetector from the computer vision toolbox), using a model of the background to generate a binary mask that highlights the foreground in white and the background in black. See Figure 73 for an example of moving object detection in the reference software. This technique is available in OpenCV, through the cv::BackgroundSubtractorMOG2 class, and has been explored in the development of the OpenCV software and was not adopted in the end (see section 3.5.5.1).

Both techniques have their advantages and disadvantages. The difference operation is much more straightforward to implement, whereas the foreground technique requires a deeper understanding of the background generation algorithm to achieve accurate results. Also, the difference operation technique is less sensitive to motion, and creates a very weak outline of moving objects. For this reason, the resulting frame must undergo a dilation operation to enhance contours and thresholding to generate a binary mask (see section 3.4.3). On the contrary, foreground detection tends to be too sensitive to even the slightest perturbations. Therefore, it requires morphological opening and closing operations as well as filling “to remove noisy pixels and to fill the holes in the remaining blobs” [40], and the resulting frame can be seen in Figure 74. Both OpenCV and MATLAB feature built-in functions for morphological operations with variable kernel sizes and shapes.

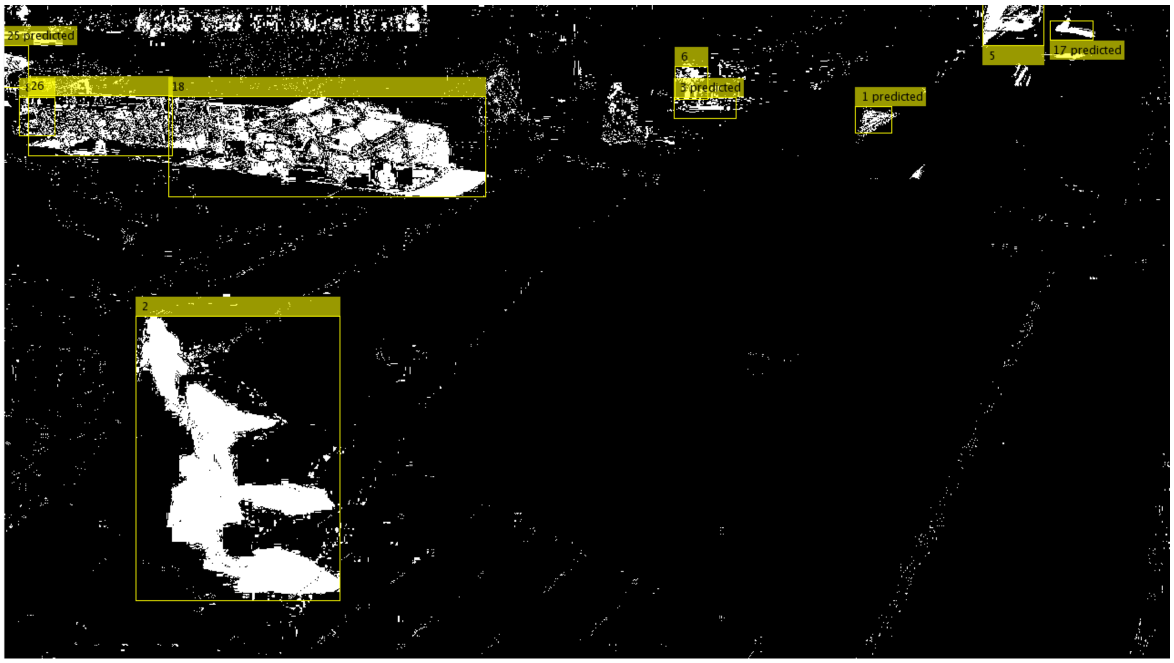


Figure 73: Moving object detection in the reference software

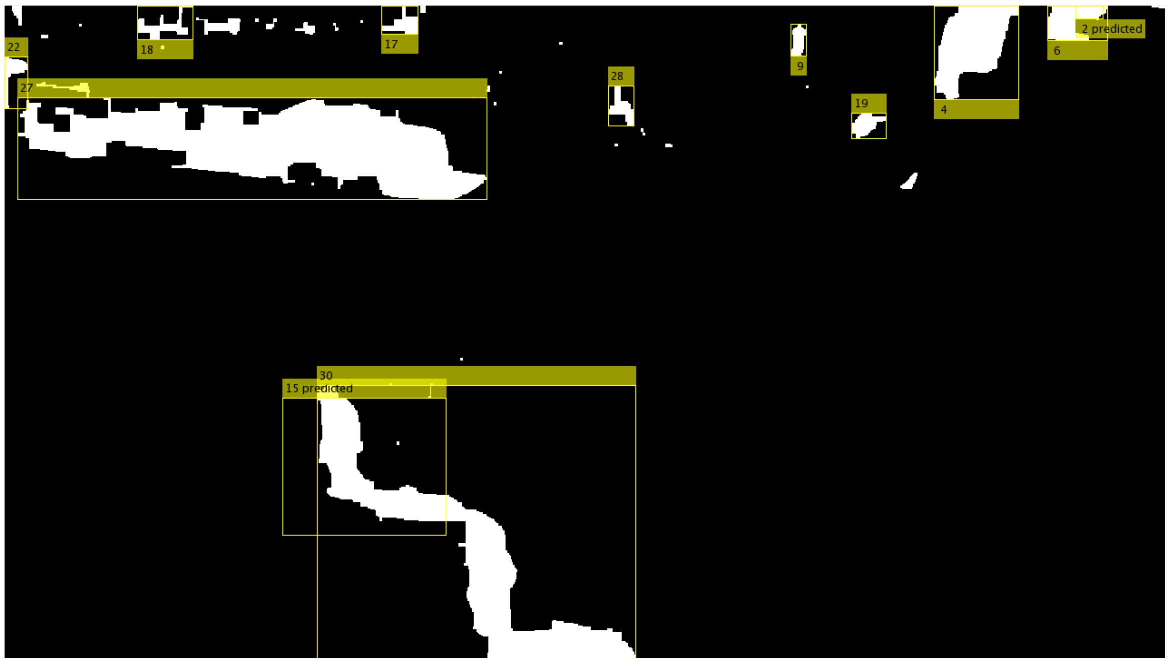


Figure 74: Moving object detection in the reference software after morphological operations

The new software features some optimizations that are not present in the reference one and enhance the accuracy of contour detection. Both motion detection techniques detailed above are vulnerable to shadows and detect them as part of objects, but only the new software has a shadow-masking technique that attempts to solve this problem (see section 3.4.2). For contour detection, the reference software has a very simple criterion for whether to select a contour as a valid object: it must have a certain minimum area, whereas the new one applies a multitude of checks. The object's area is taken into account, but the threshold varies depending on where the object is located in the frame thanks to the perspective correction (see section 3.5.2). The dimensions and convexity of the contour are also taken into account to determine whether its shape fits the required profile. Visually, the reference software displays the bounding boxes of contours, whereas the new software displays the exact contour outlines, but this is only a visual distinction. In terms of ease of implementation, the reference software is superior thanks to features in MATLAB: all relevant contour characteristics (area, centroid, rejection of small contours) are conveniently packed in the vision.BlobAnalysis component of the computer vision toolbox, while OpenCV requires individual function calls for each of the above parameters.

4.4.1.2 Object tracking

Both programs track objects in a similar way: in essence, they define each physical object as initial and unique ID value for each object, which also holds information on the (through the ObjectContour object for the new software or the track struct for the reference software). They link each detected contour to a previously detected object's contour. They then update each object (ID) with the most recent contour characteristics (contour points bounding box, centroid, etc.) and increment the object "age" or "lifetime" parameter. Neither OpenCV nor MATLAB provide functionality to perform these operations, the objects or structures must be defined manually.

The main difference between the reference and new programs lies in how each software links contours from one frame to the next. The new software has a very straightforward way of tracking objects: it links a given contour from the current frame to the one from the previous frame that has the shortest Euclidean distance to it (below a certain threshold), ensuring that each old contour is linked to a maximum of one new contour. It can also link new contours to old contours up from up to three frames behind, ensuring that tracking is not lost if a contour disappears for one or two frames, giving old contours priority. For more details, see section 3.5.3.2.

As for the reference software, it has a much more complex way of linking contours. It first converts each detected contour to a track structure, which combines it with parameters such as ID and age. On the next frame, the expected position of the contour track is computed based on its previous position and velocity using a Kalman filter provided in MATLAB's computer vision toolbox (for more information on the Kalman filter and its use in MATLAB, see [41]). Then, to actually perform the linking operation, the software solves an optimization problem. It "compute[s] the cost of assigning every detection to each track using the |distance| method of the |vision.KalmanFilter| […] The cost takes into account the Euclidean distance between the predicted centroid of the track and the centroid of the detection. It also includes the confidence of the prediction, which is maintained by the Kalman filter" [40]. Solving this problem returns a matrix containing each track and the corresponding contour from the current frame [40] and the linking operation is done. Then, the tracks which have been assigned a contour are updated with the parameters of the contour (centroid, bounding box) and the age of every track (assigned and unassigned) is increased. Finally, tracks that have not been assigned a contour for too many frames are deleted, as the object is considered to have disappeared. The unassigned tracks that are not yet deleted have their predicted position displayed to the user, unlike the new software where they are simply not shown. For an example of tracking in the reference software, including predicted track locations, see Figure 75.

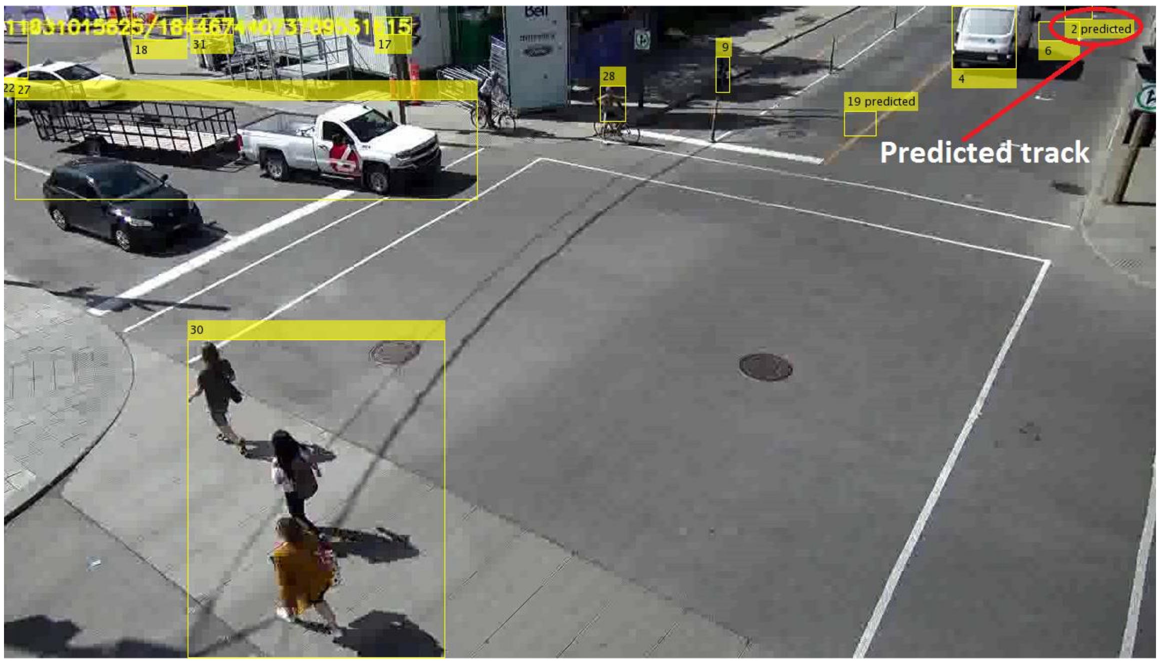


Figure 75: Object tracking in reference software

The tracking in the reference software is theoretically is more accurate than the method used in the new software, which does not account for the displacement of vehicles when calculating which contour from the new frame is closest to each previous frame's contour, and does not perform any optimization. Such a technique could be implemented in the new software, since OpenCV also implements the Kalman filter [42], but the advantages may not be significant. Indeed, the lack of prediction of vehicle position in the next frame has not been an issue, since vehicles travel relatively slowly at intersections, so it is very unlikely that the tracking algorithm wrongly assigns a vehicle's contour to another vehicle. However, the way the reference software handles unassigned tracks is superior to the new software, since it would solve a commonly occurring issue where two vehicles are detected as a single contour and one of the tracks immediately disappears.

4.4.1.3 Object counting

The new and reference programs have vastly different approaches to counting. The new software implements a counting algorithm comprising of several steps: detecting the number of new or deleted contours, determining where each entered or exited the frame, and incrementing the count if some conditions are satisfied (see section 3.5.3.3 for more details). Built-in functions for this purpose are not present in OpenCV (or MATLAB for that matter), so they had to be written from scratch. In comparison, the reference software simply displays the number of contours currently detected in the frame. This method could allow counting the number of vehicles entering and exiting the frame by subtracting the contour counts between consecutive frames, but this method is unreliable due to the possibility of vehicles stopping in the middle of the frame and not being detected.

4.4.2 Comparison of software performance and ease of development

4.4.2.1 Software performance

In general, MATLAB-developed software performs slower than the C++ OpenCV equivalent [43] [44] because MATLAB is interpreted, while C++ is compiled, which means that MATLAB code has to be converted to Java and executed during runtime, while C++ code runs directly [43]. This can result in a speed difference from 3-4 FPS (MATLAB) to 30 FPS (OpenCV) [43]. Also, OpenCV typically uses far less memory than MATLAB for computer vision: for a similar task, MATLAB may require over 1GB and OpenCV, only 70MB [43]. However, MATLAB's linear algebra routines are very fast, so it can sometimes outperform C++ code written by non-experts if linear algebra is involved, which is quite likely because images are treated as multi-dimensional matrices in computer vision [44]. Overall, if the new and reference programs implemented the exact same algorithms, the OpenCV program is expected to perform better with less resource utilization, except in some portions such as solving the optimization problem for tracking (see section 4.4.1.2).

In terms of performance for this specific application (traffic counting), it is not very accurate to compare the performance of OpenCV and MATLAB using the two programs, since they do not perform the same operations (the detection is slightly more complex in the new software, the tracking is more complex in the reference one, and the counting is significantly more complex in the new software). Nevertheless, the most resource-intensive operations are expected to be the image processing steps, which are similar between the two programs, so the comparison is not totally invalid. When collecting performance data, the test clip was Example 1 from the motion detection video clips (see section 4.2.1), whose performance, CPU utilization, and RAM utilization were found in section 4.3.2.2. When running the OpenCV-based new software , CPU usage was between 27% and 38%, with an average of 31.75%. As for RAM, the software required between 161.1 MB and 170.6 MB, with an average of 166.73 MB. With this resource usage, the software was able to run in real-time (5 minutes to process a 5-minute clip), although it only truly processed one-third of the frames (the rest being used to compute the difference frame only). As for the MATLAB-based reference software, CPU usage was between 48% and 57%, with an average of 50.93%, which is nearly 60% more than OpenCV. It also used a constant 1.6 GB of RAM, a whole order of magnitude more than OpenCV requires. With this resource utilization, the software took minutes 14.82 minutes to analyze the 5-minute clip, representing a frame rate of 8.434 FPS and only 33.74% of real-time performance. This discrepancy between the percentage of real-time operation between the two programs is mainly due to the fact that the reference software analyzes all frames and the new one does only one-third, so it has an effective FPS of 8.325 FPS. However, even taking that into account, the new software is more efficient than the reference software: it achieves essentially the same frame rate while having lower resource utilization and performing additional image processing steps that are not done by the reference software, such as the shadow mask operation. Therefore, one can conclude that OpenCV-based software will generally perform better than a MATLAB equivalent.

4.4.2.2 Ease of development

In general, MATLAB development can be considered to be easier than OpenCV, for many different reasons. MATLAB is a scripting language, while OpenCV is used with C++, a programming language, so code that performs the same function is going to be much shorter and more readable in MATLAB than OpenCV [43]. Furthermore, C++ is one of the most difficult programming languages because it requires memory management [43] [44]. Indeed, during the development of this project, many compilation and runtime errors occurred, and took a long time to diagnose and solve due to the nature of C++. It is much more convenient to learn and use MATLAB than OpenCV, although one must be careful to write code "the MATLAB way" [44] to achieve better performance. It must also be noted that OpenCV can be used with Python, which nullifies many of the disadvantages of C++, but also its main advantage, speed [44]. Therefore, this alternative cannot be recommended.

Also, MATLAB provides a very complete and user-friendly development environment, which provides more convenient visualization of results, and especially much easier debugging [44]. In comparison, for C++, the programmer has the choice of several development environments [43], but neither quite matches the simplicity and completeness of the MATLAB one. For example, in MATLAB, the exact line where a syntax error or bug occurs is indicated, with a clear description of the error. In comparison, in C++ and OpenCV, syntax errors often generate lines upon lines of cryptic messages, and there is absolutely no indication as to where runtime errors occur, except the dreaded “segmentation fault” or “assertion failed” messages.

MATLAB also has better documentation than OpenCV [44], which is very useful for non-experts in computer vision. Although almost all OpenCV features are documented, the documentation has fewer tutorials and less sample code than MATLAB’s and sometimes a very poor explanation of the effect of each parameter [44]. This has led to countless hours spent doing trial and error during the development of the OpenCV-based software, and using MATLAB would likely have reduced that somewhat.

Finally, the availability of library functions can play a significant role in ease of use: if one development package provides a certain required function and not the other, using the former will greatly simplify the task. As seen in section 4.4.1, for this project, neither OpenCV nor MATLAB have a function that the other one does not implement, but MATLAB’s implementations are generally easier to use. For future work, it must be noted that OpenCV has a much larger library of computer vision functions, but MATLAB has a superior machine learning library. Therefore, the choice of software will depend on the direction the project takes (continue working with traditional computer vision or move on to machine learning).

4.4.3 Comparison of software accuracy

Since the reference software does not support counting in the same way as the new one, a quantitative comparison is not possible, so a qualitative comparison of the performance of vehicle detection and tracking will be provided. Assuming that the detection and tracking of the reference software can perform similarly to the new one, it is expected that the counting algorithm would produce the same results if it is implemented in the reference software. However, it would perform somewhat slower in light of the findings in section 4.4.2.1.

4.4.3.1 Qualitative comparison of object detection

In terms of detection, the reference software does not perform nearly as good as the new one, for several reasons. Firstly, the moving object detection using foreground detection is not ideal in an urban intersection environment, where the background changes constantly (for example, cars that are stopped at a traffic light are detected as part of the generated background). Therefore, the background must be regenerated every few seconds. But every time a new background is generated, the mask is completely corrupted for a few seconds: most of it becomes white, instead of only the vehicles (see Figure 76). This is a similar issue to the one experienced when foreground detection was attempted in the new software (see section 3.5.5.1), which led to its rejection. Secondly, the morphological operations are implemented rather poorly in the reference software. The opening operation, which is supposed to remove small regions of pixel noise, uses a kernel size that is too small and noise remains in the final detection mask. The closing operation, which is supposed to close small holes in objects, also uses a smaller-than-optimal kernel size, so objects are often detected as multiple contours. Also, the rectangular kernel chosen makes contours have rough rectangular edges (see Figure 74), which is why the elliptical kernel was chosen when developing the new software (see section 3.4.3). Thirdly, there is no perspective compensation at all, only a minimum contour area. This causes issues where too many small contours (glitches) at the bottom of the frame (where objects appear largest) are above the threshold area and are detected, and where some actual vehicle contours at the top of the frame (where objects appear smallest) are below the threshold area and are rejected. Figure 77 shows examples of actual contours not detected while glitches are detected.

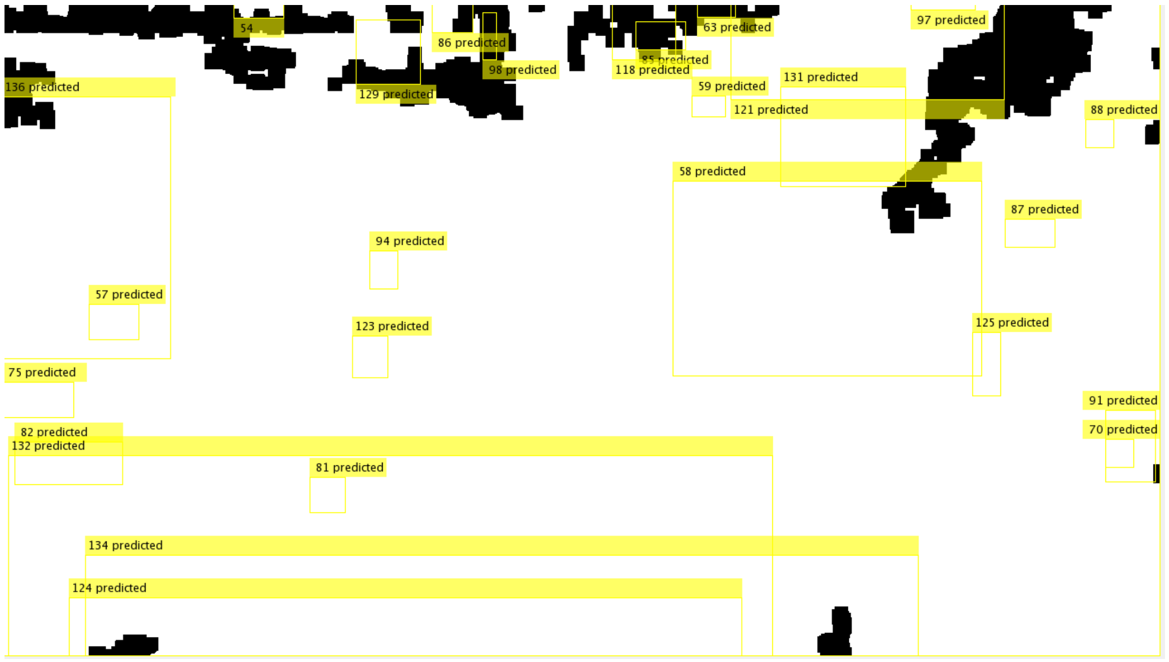


Figure 76: Corrupted foreground mask in reference software when background model refreshes

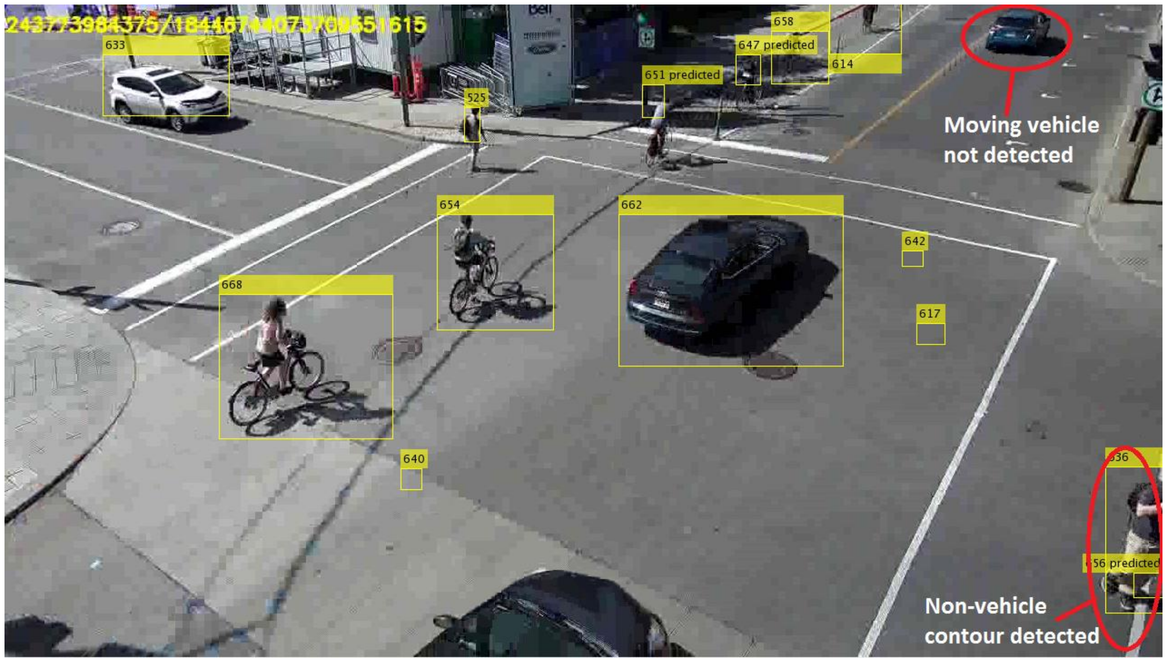


Figure 77: Glitches detected while actual contours not detected in reference software due to lack of perspective correction

4.4.3.2 Qualitative comparison of object tracking

As for the tracking performance of the reference software, it appears to be better than the new software, though it is hard to tell because of all the poor detections inducing errors in tracking. Nevertheless, a few advantages over the new software can be seen. When a contour briefly disappears and reappears, the new software does not show the contour in the frame where it is not detected, but the reference software will show the predicted location of the disappeared contour, providing a better visual feedback and reducing the likelihood of the track being lost (due to the matching being done based on the predicted location). Furthermore, in the new software, when two contours merge briefly (for a few frames), one of the contours will disappear, and when they separate again, it is likely that the tracking will be broken for at least one of them, or worse, that the ID value of one of the contours will be assigned to the other. In comparison, in the reference software, they will remain as two separate tracks and no detection error should occur.

5. Conclusion

5.1 Parking detection

5.1.1 Camera placement suggestions

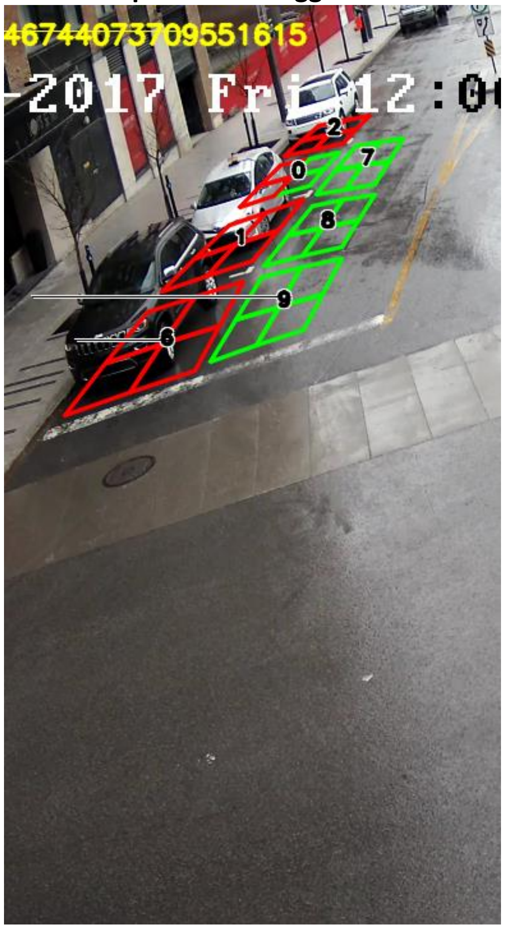


Figure 78: Parking detection issue due to inadequate perspective (zone 0)

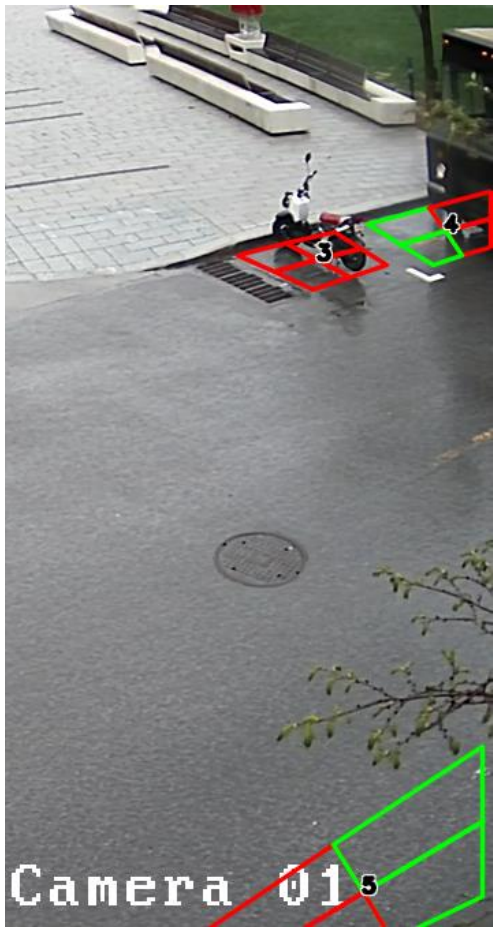


Figure 79: Parking detection issue due to inadequate framing (zone 4) and text

Some difficulties in parking detection are caused by the position of the camera and its perspective. For example, the fact that the camera has a perspective view causes issues with vehicles blocking the view of parking zones behind them (for example, zone 0 is blocked by 1, 2 is blocked by 0…). This has been

partially remedied by the multiple polygons per zone (see section 3.3.1). However, a sufficiently tall vehicle or one that is parked too close to the edge of a given spot will still cover two or three polygons of the adjacent zones, leading to a false detection: despite efforts to resolve this issue in software, it still occurs. Another problem related to the camera position is that parking spots 4 and 5 are not entirely visible. Therefore, it can be difficult for the detection algorithms to determine whether the spot is occupied or free if the vehicle is positioned such that most of it is outside the frame and only a small portion of it is visible. The software solution to this is simply to reduce the edge detection threshold for partially visible zones so they are marked as occupied if a small portion of the vehicle covers them. However, lowering the threshold too much leads to glitches where an obviously empty spot is recognized as free due to irregularities of the asphalt, so this solution also has its limitations. The following images show examples of wrong detections caused by suboptimal camera position:
In **Figure 78**, the car in zone 1 is parked very close to the edge between zones 1 and 0, so its rear covers two out of four polygons in zone 0. Therefore, zone 0 could potentially be recognized as occupied while it is in fact free, since a transition to 2/4 free zones does not result in a change of state (see section 3.3.1).
As for Figure 79, it demonstrates the issue of camera framing not including the entirety of parking zone 4: although a vehicle is present, only two out of four polygons are marked as occupied, so it could potentially be recognized as free by the software.
Another issue caused by the camera is the presence of the camera name and timestamp that are baked into the video stream. They cover some regions of interest, such as zone 5, and their presence causes errors in the parking detection. For example, in Figure 79, the top left corner is falsely marked as occupied due to the edges from the camera name which contrasts with the pavement. Removing the name and timestamp automatically using OpenCV would be extremely difficult to implement, computationally expensive, and inaccurate.

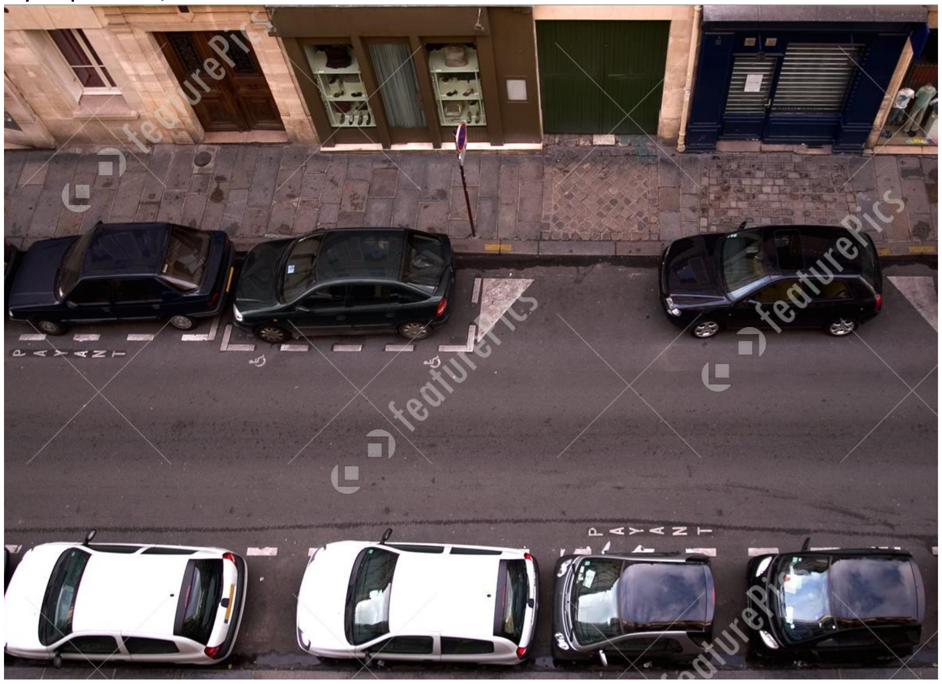


Figure 80: Ideal camera setup for parking detection

Source: http://www.featurepics.com/StockImage/20070709/street-parking-paris-stock-picture-374677.jpg

In light of the above findings, the optimal camera position satisfies the following constraints. The most important factor is the camera angle: it is positioned so that it has an approximately top-down view to solve the perspective issue from Figure 78. It also has framing such that every parking spot to be analyzed is fully visible to solve the issue from Figure 79. Finally, it should not have a timestamp that interferes with the visual content of the frame. Another study on stationary vehicle detection (i.e. parking detection) confirms that camera angle is critical for this application and should be as high as possible, since low camera angles lead to detection issues due to the tridimensional nature of vehicles, but also the presence of shadows [45]. An example of ideal camera position taking into account the above factors is shown in Figure 80 below:

5.1.2 Objective achievement

After the greater part of a month was spent developing and testing the parking detection software, using footage from various times of day and weather conditions (except snow, which was impossible), it was determined that most of the objectives that were set were achieved. First, the software should be able to determine which parking spots and illegal parking zones are occupied or free, which of course comes with a certain reliability standard. During testing, in typical conditions, it was found that the reliability was very high, above 90% in all cases, and nearly 100% on average (see section 4.1.1). The software is indeed capable of analyzing detection results from multiple polygons to accurately determine the parking status of a given spot, which it can write to a database (see section 3.1.2). Therefore, this objective is considered to be achieved. Then, the program should function autonomously after an initial set up process to minimize use of human resources. The current iteration of the program functions across a range of light and weather conditions using a single set of parking zone data and image processing parameters that are defined in an initial phase and require no modification unless the camera position changes, which does not typically occur. Thus, this objective is also achieved. Finally, the software should provide reliable detection in non-ideal conditions such as low light and poor weather conditions. Although several features were added to deal with difficult conditions, such as histogram equalization for night (see section 3.2.2) and denoising for rain (see section 3.2.3), there are still some difficulties in parking detection at night, with parking status not being detected properly in several zones in the middle of the night (see section 4.1.2). Unfortunately, this third and last objective is still not fully achieved.

5.1.3 Future work

There are still improvements that could be done to the parking detection software. Most importantly, the reliability of parking detection during night-time must be improved until the success rate approaches that of daytime. This could be done through tweaking parameters further, or by adding/modifying parking detection algorithms. Also, one could develop an auxiliary piece of software that parses the parking event log file, cross-references it with a database indicating the rules of parking (which zone is open for parking at which times, how long can a given car park at a certain spot), and issue alerts when illegal parking is detected. The user interface could also be improved: it still requires the use of command line and editing text files, while a stand-alone executable file and graphical user interface for setting all parameters (video stream, settings)… could also be added.

5.2 Motion detection

5.2.1 Camera placement suggestions

Throughout the entire testing phase of the motion detection software, a major difficulty was the camera placement. In the initial video steam (intersection of Clarke and Ontario, 172.168.10.30), the camera was placed very poorly, as shown in Figure 26 (section 3.4.1). The bottom of the frame, the part of the scene where it is easiest to distinguish objects, does not have any vehicles entering/exiting! The camera is also tilted, so vehicles entering the frame from the left often appear at the middle bottom part of the frame. Furthermore, traffic detection at the top of the frame has many issues. The part where vehicles are the most difficult to distinguish due to perspective) has a lot of traffic (vehicles both entering and exiting), so many vehicles fail to be counted. Also, a small portion of another intersection is (barely) visible, so many vehicles that pass through this intersection are not counted properly (often marked as “exited” without being marked as “entered” first). The second video stream (intersection of Jeanne-Mance and President-Kennedy, 172.168.10.32) is significantly better, since the camera is adjustable using pan, tilt, zoom

controls. However, the optimal configuration that was found, shown in Figure 8 (section 3.1.3), still has a few flaws. In order for the top of the frame to be somewhat visible, the camera had to be tilted so that the bottom of the frame is barely visible. Therefore, a small fraction of vehicles that enter from the bottom and make a right turn fail to be detected because they only partially enter the frame. Furthermore, the same problem occurs with the top part of the frame: vehicles are too small and it can be difficult to distinguish them if they follow each other. Considering these findings, the following camera placements are suggested depending on the availability of hardware and the priorities of the application.
For applications that focus purely on obtaining the most accurate counts, the best results would be achieved with one camera recording each street and only detecting objects that pass directly underneath, as in Figure 81. This would provide the most accurate detection of individual objects, since it focuses on the most easily visible ones. It would also require no perspective compensation, since all objects would be detected in the same area of the frame. However, this technique requires four cameras (for each street), which is a significant investment and is against the principles of multi-purpose cameras pursued in this project. Furthermore, since video footage comes from four different cameras, tracking an object across the whole frame and thus obtaining statistics on the trajectory of objects will not be supported.
If only one camera is available, the best view would be to look at the entire intersection from the top, as in Figure 82. This would have the highest possible view angle, which is suggested in other research on this topic because it "limits the occlusion between densely spacely vehicles" [46], which would reduce the number of multiple vehicles being combined in a single contour. This solves the problem of vehicles overlapping and being difficult to identify, and should minimize the impact of shadows. It also requires no perspective adjustment, since there is no perspective issue, and the distortion introduced by the fisheye effect is minimal. Furthermore, if both parking and motion detection are to be done with the same camera, this perspective will work well for both. However, mounting the camera directly above the intersection may not be possible in urban environments, where cameras are most often mounted with a low camera angle [46]. Also, detecting pedestrians with this perspective will be very tricky because of their very vertical shape: their contour area will vary significantly based on whether they are directly below the camera, or closer to the edge of the frame.

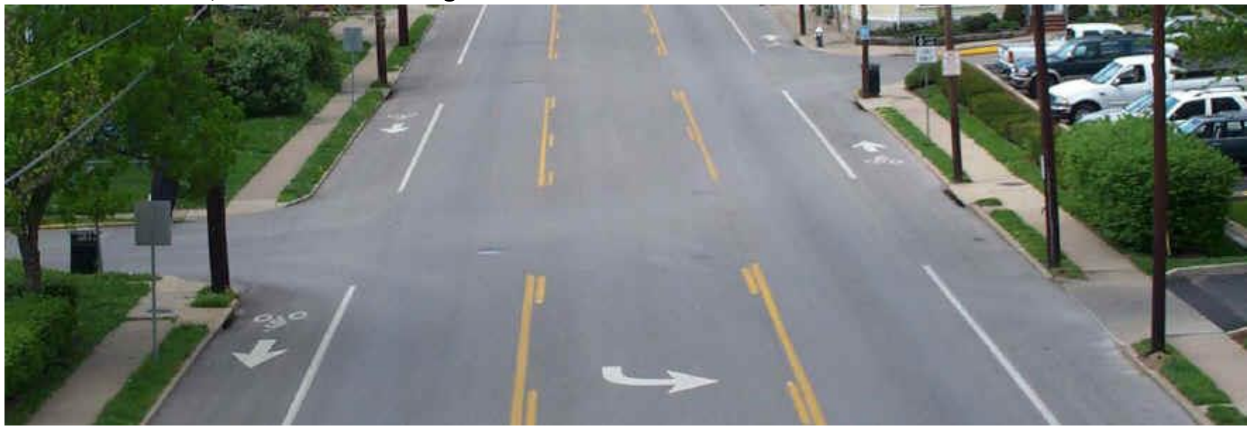


Figure 81: Camera looking down a street

Source: http://transportation.ky.gov/Congestion-Toolbox/PublishingImages/RoadDiets.png

The next suggestion would be an isometric view, similar to Figure 83. This method is suggested because it does not involve looking down a street, so it avoids the problem of having multiple vehicles detected as one at the end of the street, where detected vehicles are small and hard to distinguish. It also requires little perspective compensation compared to looking down at a street directly. Another significant advantage is the fact that all lanes of each street should be visible, unlike a perspective where one of the streets goes across the frame. It supports pedestrian detection and should also be the most feasible camera perspective to set up in an urban area and to use for multiple purposes. Despite these advantages, accuracy is expected to be worse than the above two suggestions, so it should only be considered when the functionality advantages it offers are crucial, such as in this project where it was used. The following PTZ settings are the closest the camera at Jeanne-Mance and President-Kennedy can be to an isometric view: pan 37.9°, tilt -28.6°, zoom 1x (ideally the camera would be mounted higher and a tilt of -45° would be used).

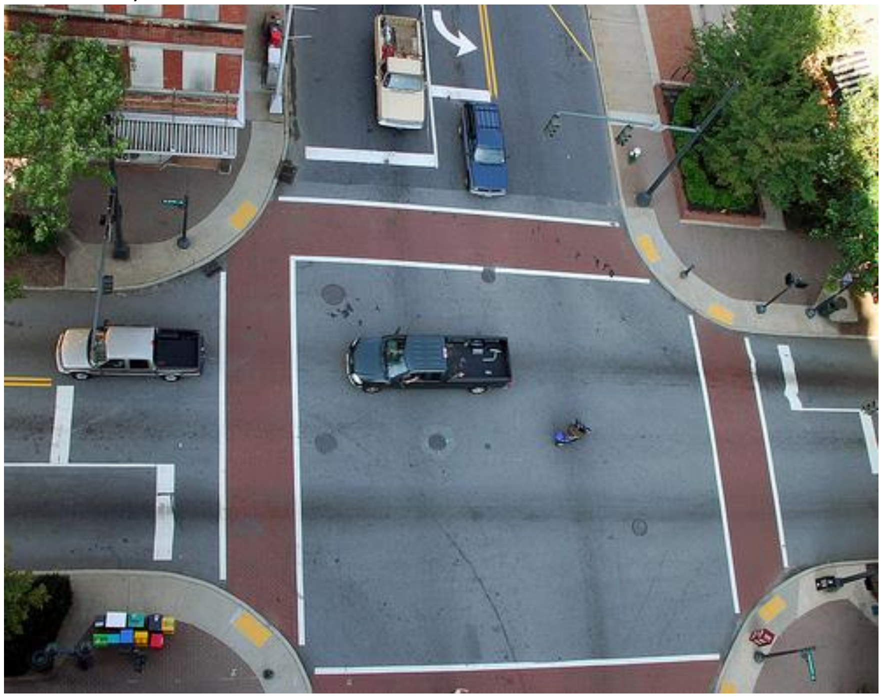


Figure 82: Top-down view

Source: http://farm4.static.flickr.com/3497/3236435965_47f187150a.jpg

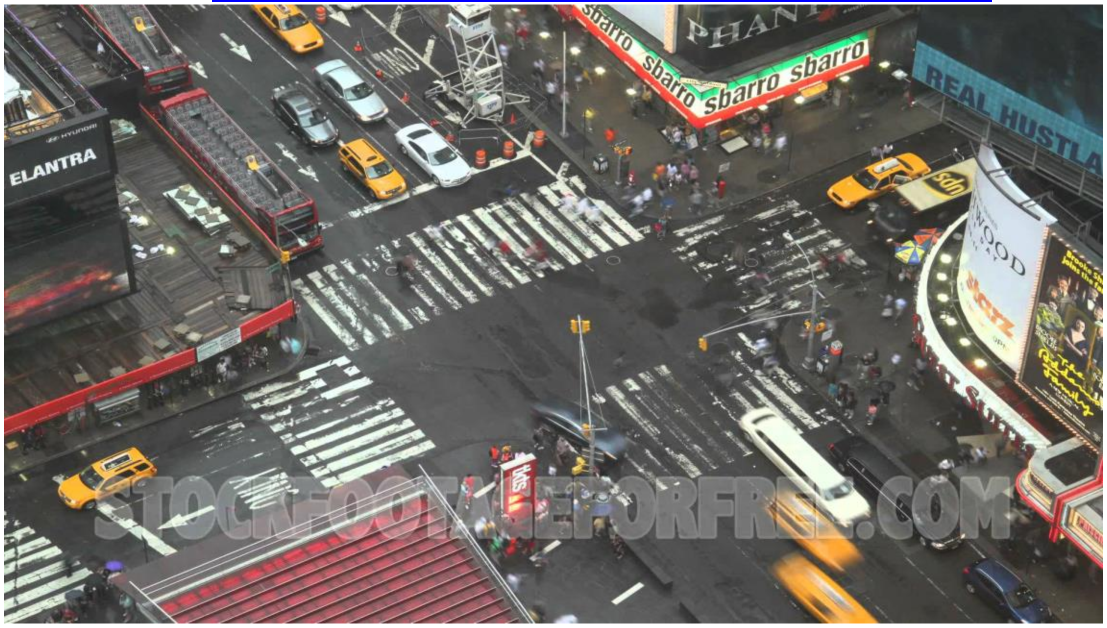


Figure 83: Isometric view

Source: https://i.ytimg.com/vi/HoY8kG2DosA/maxresdefault.jpg

5.2.2 Objective achievement

After developing and testing this software for more than a month, it was determined that the traffic detection objectives were partially achieved. First, the software should be able to detect and distinguish pedestrians and vehicles of various classes. This requirement is not entirely satisfied, since the software currently only detects vehicles (approximate size and speed of a car), and pedestrians. Cyclists and motorcyclists are too small to count as vehicles and too large and/or too fast to count as pedestrians, so they are most often ignored (except for a few occasional glitches where they are counted as one or the other). The software should also track the detected objects as they pass through the area and count the number of them in each direction. This requirement is satisfied: these algorithms work as intended, and counts are very accurate (90% and above). The requirement that the software functions autonomously after an initial setup process is also satisfied. However, the motion detection software should also provide reliable detection even in non-ideal conditions, but it performs somewhat worse in some conditions. Indeed, since the shadow mask has temporarily been disabled, shadows are a big problem in the morning and evening: they cause multiple vehicle contours to be counted as one, lowering accuracy and consistency (the worst results in terms of both metrics are from Example 1, the morning clip).

5.2.3 Future work

5.2.3.1 Shadow mask improvements

The current iteration of the shadow mask generation algorithm has some flaws. The main issue is that the color characteristics of shadows change during the day: in the morning and afternoon, shadows are much lighter than at midday. Therefore, shadow detection thresholds (section 3.4.2) configured for the morning are completely ineffective during the day. This should not be a major problem because shadows are only very problematic in the morning, when they are the longest and cause multiple contours to be combined into one. However, the fact is that the color profile of shadows is too similar to that of black vehicles and causes some of them to be masked out as well. Therefore, for most of the day, using the shadow removal algorithm has more disadvantages than advantages.

One way to solve these problems would be to implement a second set of conditions for determining the location of shadows that is based on a different characteristic than chromacity. A characteristic of shadows is that they are comprised of few edges compared to objects of interest, so the shadow mask should include only areas with few edges. One attempt that was made used the OpenCV's Laplacian function cv::Laplacian() described in section 3.2.2 that detects edges of objects, but it was ineffective because it also detected the edge of the shadow, and that is sufficient for the entire shadow to be added to the object's contour. A similar approach was used with success in [47], where color and texture (edge, corner) information were used in conjunction to separate shadows and black vehicles accurately. The difference is that they used the CIE L*a*b* color space instead of HSV, and the SUSAN mask instead of the Laplacian [47]. The SUSAN mask represents the following operations: for each pixel $I(x_0, y_0)$ and all neighboring pixels in a 7x7 grid $I(x, y)$, compute $|I(x, y) - I(x_0, y_0)|$, the absolute difference, then let $E(x, y) = 1$ if $|I(x, y) - I(x_0, y_0)| \leq T$ (threshold value) and 0 otherwise [47]. Then, for each pixel $I(x_0, y_0)$, calculate $N(x_0, y_0) = \sum_{i=1}^{7} E(x_i, y_i)$ where N is the total number of pixels centered at (x_i, y_j), and define $R(x_0, y_0) = 0$ if $N(x_0, y_0) < g$ (threshold value equal to $\frac{3}{4} \cdot N_{max}$) and 1 otherwise (a value of 1 corresponds to a shadow) [47]. Linear filtering is also performed to remove the edge of the shadow, a problem that was also encountered [47].

5.2.3.2 Feature-based detection

Most of the issues that arise in identifying the proper contours (vehicles, pedestrians) when using this software are due to the fact that only the contours are considered, not features that characterize the objects in question. Adding feature-based detection would help separate pedestrians, cyclists, and vehicles much more accurately than any algorithm that has been devised so far (based on shape dimensions). OpenCV implements a Haar feature-based cascade classifier [48], which will be explored as part of the next phase of the project. The disadvantage of this method is that it requires training, with a few hundred positive samples that contain the object of interest and negative samples that do not [48]. However, once this is done, the Haar classifier is easy to use: select a region of interest in the image, and the classifier will output “1” if it contains the object, and “0” if it does not [48]. It could even be used in conjunction with motion detection: the difference frame could be used to mask the original frame, and the cascade classifier would be applied on the resulting frame, so that it considers only moving objects. The next technical report will contain more details about the Haar classifier and potentially other similar image detection methods applied to the detection and counting of pedestrians and vehicles as part of the SmartCity project.

References

[1] Nicholas True. “Vacant parking space detection in static images,” University of California, San Diego, 17, May 2007.

[2] Sangwon Lee, Dukhee Yoon, and Amitabha Ghosh. "Intelligent parking lot application using wireless sensor networks." *Collaborative Technologies and Systems, 2008. CTS 2008. International Symposium on*. IEEE, 2008.

[3] Kunfeng Wang, et al. "An automated vehicle counting system for traffic surveillance." *Vehicular Electronics and Safety, 2007. ICVES. IEEE International Conference on*. IEEE, 2007.

[4] Guillaume Leduc. "Road traffic data: Collection methods and applications." *Working Papers on Energy, Transport and Climate Change* 1.55 (2008).

[5] Yoichiro Iwasaki, Masato Misumi, and Toshiyuki Nakamiya. "Robust vehicle detection under various environmental conditions using an infrared thermal camera and its application to road traffic flow monitoring." *Sensors* 13.6 (2013): 7756-7773.

[6] GitHub (user eladj). (2016, Nov 24). *Automatic Parking* Detection [Online]. Available: https://github.com/eladj/detectParking

[7] GitHub (user andrewssobral). (2017, Apr 12). *Vehicle Detection, Tracking and Counting* [Online]. Available: https://github.com/andrewssobral/simple_vehicle_counting

[8] Hardik Madhu. (2013, Sep 27). *Motion Detection and Speed Estimation using OpenCV* [Online]. Available: http://hardikmadhu.blogspot.ca/2013/09/a-simple-and-minimal-aproach-to-human.html

[9] Adrian Rosebrock. (2015, May 25). *Basic motion detection and tracking with Python and OpenCV* [Online]. Available: http://www.pyimagesearch.com/2015/05/25/basic-motion-detection-and-tracking-with-python-and-opencv/

[10] Matthias Stein. (n.d.). *Motion detection using a webcam, Python, OpenCV and Differential Images* [Online]. Available: http://www.steinm.com/blog/motion-detection-webcam-python-opencv-differential-images/

[11] OpenCV. (2015, Dec 18). *Color conversions* [Online]. Available: http://docs.opencv.org/3.1.0/de/d25/imgproc_color_conversions.html

[12] OpenCV. (2014, Nov 10). *Image Filtering* [Online]. Available: http://docs.opencv.org/3.0-beta/modules/imgproc/doc/filtering.html

[13] OpenCV. (2015, Dec 18). *Histograms - 2: Histogram Equalization* [Online]. Available: http://docs.opencv.org/3.1.0/d5/daf/tutorial_py_histogram_equalization.html

[14] Stephen M. Pizer et al. “Adaptive histogram equalization and its variations.” *Computer vision, graphics, and image processing*, vol. 39, no. 3, pp. 355-368, 1987.

[15] OpenCV. (2017, May 25). *Image denoising* [Online]. Available: http://docs.opencv.org/trunk/d5/d69/tutorial_py_non_local_means.html

[16] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. “Non-local means denoising.” *Image Processing On Line*, vol. 1, pp. 208-212, 2011.

[17] OpenCV. (2015, Dec 18). *Laplace Operator* [Online]. Available: http://docs.opencv.org/3.1.0/d5/db5/tutorial_laplace_operator.html

[18] OpenCV. (2017, May 25). *Changing the contrast and brightness of an image!* [Online]. Available: http://docs.opencv.org/trunk/d3/dc1/tutorial_basic_linear_transform.html

[19] OpenCV. (2017, May 26). *Canny Edge Detector* [Online]. Available: http://docs.opencv.org/2.4/modules/imgproc/doc/feature_detection.html?highlight=canny#canny

[20] OpenCV. (2017, Jun 08). *Creating Bounding boxes and circles for contours* [Online].

Available: http://docs.opencv.org/2.4/doc/tutorials/imgproc/shapedescriptors/bounding_rects_circles/bounding_rects_circles.html

[21] Andres Sanin, Conrad Sanderson, and Brian C. Lovell. “Shadow detection: A survey and comparative evaluation of recent methods.” *Pattern recognition*, vol. 45, no. 4, pp. 1684-1695, 2012.

[22] Clément Fredembach, and Graham Finlayson. “Hamiltonian path-based shadow removal.” *Proc. Of the 16th British Machine Vision Conference*, vol. 2, no. LCAV-CONF-2017-020, pp. 502-511, 2005.

[23] Rita Cucchiara, et al. “Detecting moving objects, ghosts, and shadows in video streams.” *IEEE transactions on pattern analysis and machine intelligence*, vol. 25, no. 10, pp. 1337 1342, 2003.

[24] OpenCV. (2017, Jun 08). *cv::Background SubtractorMOG2 Class Reference* [Online]. Available: http://docs.opencv.org/trunk/d7/d7b/classcv_1_1BackgroundSubtractorMOG2.html

[25] Zoran Zivkovic. “Improved adaptive Gaussian mixture model for background subtraction.” *Pattern recognition,* Proceedings of the 17th International Conference on Pattern recognition, 2004. ICPR 2004., vol. 2, pp. 28-31, IEEE, 2004.

[26] OpenCV. (2016, Dec 23). *How to Use Background Subtraction Methods* [Online]. Available: http://docs.opencv.org/3.2.0/d1/dc5/tutorial_background_subtraction.html

[27] OpenCV. (2017, Jun 8). *Motion Analysis and Object Tracking* [Online]. Available: http://docs.opencv.org/2.4/modules/video/doc/motion_analysis_and_object_tracking.html

[28] John W. Shipman. (2012, Oct 16). *The hue-saturation-value (HSV) color model* [Online]. Available: http://infohost.nmt.edu/tcc/help/pubs/colortheory/web/hsv.html

[29] OpenCV. (2014, Nov. 10). *Morphological transformations* [Online]. Available: http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html

[30] OpenCV. (2017, Jun 08). *Eroding and Dilating* [Online]. Available: http://docs.opencv.org/2.4/doc/tutorials/imgproc/erosion_dilatation/erosion_dilatation.html

[31] StackOverflow (user Martin Beckett). *OpenCV: Matrix Iteration* [Online]. Available: https://stackoverflow.com/questions/11977954/opencv-matrix-iteration

[32] OpenCV. (2017, Jun 08). *Image Thresholding* [Online] . Available: http://docs.opencv.org/trunk/d7/d4d/tutorial_py_thresholding.html

[33] OpenCV. (2017, Jun 08). *Structural Analysis and Shape Descriptors* [Online]. Available: http://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html

[34] C-H. Teh, and Roland T. Chin. “On the detection of dominant points on digital curves.” *IEEE Transactions on pattern analysis and machine intelligence*, vol. 11, no. 8, pp. 859-872, 1989.

[35] David H. Douglas, and Thomas K. Peucker. “Algorithms for the reduction of the number of points required to represent a digitized line or its caricature.” *Cartographica: The International Journal for Geographic Information and Geovisualization*, vol. 10, no. 2, pp. 112-122, 1973.

[36] Andrew Kirillov. (2007, Mar 27). *Motion Detection Algorithms* [Online]. Available: https://www.codeproject.com/Articles/10248/Motion-Detection-Algorithms

[37] Charles Poynton. (1997, June 19). Frequently Asked Questions about Color [Online]. Available: http://www.poynton.com/PDFs/ColorFAQ.pdf

[38] OpenCV. (2015, Dec 18). *Color conversions* [Online]. Available: http://docs.opencv.org/3.1.0/de/d25/imgproc_color_conversions.html

[39] Ming-Kuei Hu. “Visual pattern recognition by moment invariants.” *IRE transactions on information theory*, vol. 8, no. 2, pp. 179-187, 1962.

[40] MATLAB. (n.d.) *Motion-Based Multiple Object Tracking* [Online]. Available: https://www.mathworks.com/help/vision/examples/motion-based-multiple-object-tracking.html

[41] Mohinder S. Grewal, "Kalman filtering." *International Encyclopedia of Statistical Science*. Springer Berlin Heidelberg, pp. 705-708, 2011.

[42] OpenCV. (2017, Aug 17). *cv:KalmanFilter Class Reference* [Online]. Available: http://docs.opencv.org/trunk/dd/d6a/classcv_1_1KalmanFilter.html

[43] Karan Jitendra Thakkar. (2012, Nov 21). *What is OpenCV? OpenCV vs. MATLAB —An insight* [Online]. Available: https://karanjthakkar.wordpress.com/2012/11/21/what-is-opencv-opencv-vs-matlab/

[44] Satya Mallick. (2015, Oct 30). *OpenCV (C++ vs Python) vs MATLAB for Computer Vision* [Online]. Available: https://www.learnopencv.com/opencv-c-vs-python-vs-matlab-for-computer-vision/

[45] Hankyu Moon, Rama Chellappa, and Azriel Rosenfeld. "Performance analysis of a simple vehicle detection algorithm." *Image and Vision Computing*, vol. 20, no. 1, pp. 1-13, 2002.

[46] Norbert Buch, Sergio A. Velastin, and James Orwell. "A review of computer vision techniques for the analysis of urban traffic." *IEEE Transactions on Intelligent Transportation Systems*, vol. 12, no. 3, pp. 920-939, 2011.

[47] Haiying Zhang, Qirong Zheng, and Guiwen Zheng. "A New Shadow Removal Algorithm Based on Susan and CIELAB Color Space." *Proceedings of International Conference on Internet Multimedia Computing and Service*, ACM, 2014.

[48] OpenCV. (2017, Jun 08). *Cascade Classification* [Online]. Available: http://docs.opencv.org/2.4/modules/objdetect/doc/cascade_classification.html