



Research

Algorithms for Vehicle Classification



Minnesota Local
Road Research
Board

Technical Report Documentation Page

1. Report No. MN/RC – 2000-27	2.	3. Recipient's Accession No.	
4. Title and Subtitle ALGORITHMS FOR VEHICLE CLASSIFICATION		5. Report Date July 2000	
		6.	
7. Author(s) Surendra Gupte Nikos Papanikolopoulos		8. Performing Organization Report No.	
9. Performing Organization Name and Address University of Minnesota Dept. of Computer Science and Engineering Artificial Intelligence, Robotics and Vision Laboratory Minneapolis, MN 55455		10. Project/Task/Work Unit No.	
		11. Contract (C) or Grant (G) No. c) 74708 wo) 88	
12. Sponsoring Organization Name and Address Minnesota Department of Transportation 395 John Ireland Boulevard Mail Stop 330 St. Paul, Minnesota 55155		13. Type of Report and Period Covered Final Report 1999-2000	
		14. Sponsoring Agency Code	
15. Supplementary Notes			
16. Abstract (Limit: 200 words) This report presents algorithms for vision-based detection and classification of vehicles in modeled at rectangular patches with certain dynamic behavior. The proposed method is based on the establishment of correspondences among blobs and vehicles, as the vehicles move through the image sequence. The system can classify vehicles into two categories, trucks and non-tucks, based on the dimensions of the vehicles. In addition to the category of each vehicle, the system calculates the velocities of the vehicles and generates counts of vehicles in each lane over a user-specified time interval, the total count of each type of vehicle, and the average velocity of each lane during this interval.			
17. Document Analysis/Descriptors Vehicle classification Vehicle tracking Vision-based traffic detection		18. Availability Statement No restrictions. Document available from: National Technical Information Services Springfield, Virginia 22161	
19. Security Class (this report) Unclassified	20. Security Class (this page) Unclassified	21. No. of Pages 107	22. Price

Algorithms for Vehicle Classification

Final Report

Prepared by:
Surendra Gupte
Nikolaos P. Papanikolopoulos

Artificial Intelligence, Robotics and Vision Laboratory
Department of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455

July 2000

Published by:
Minnesota Department of Transportation
Office of Research Services
First Floor
395 John Ireland Boulevard, MS 330
St Paul, MN 55155

The contents of this report reflect the views of the authors who are responsible for the facts and accuracy of the data presented herein. The contents do not necessarily reflect the views or policies of the Minnesota Department of Transportation at the time of publication. This report does not constitute a standard, specification, or regulation.

The authors and the Minnesota Department of Transportation do not endorse products or manufacturers. Trade or manufacturers' names appear herein solely because they are considered essential to this report.

Executive Summary

This report presents algorithms for vision-based detection and classification of vehicles in monocular image sequences of traffic scenes recorded by a stationary camera. Processing is done at three levels: raw images, blob level and vehicle level. Vehicles are modeled as rectangular patches with certain dynamic behavior. The proposed method is based on the establishment of correspondences among blobs and vehicles, as the vehicles move through the image sequence.

The system can classify vehicles into two categories, trucks and non-trucks, based on the dimensions of the vehicles. In addition to the category of each vehicle, the system calculates the velocities of the vehicles, generates counts of vehicles in each lane over a user-specified time interval, the total count of each type of vehicle and the average velocity of each lane during this interval. The system requires no initialization, setup or manual supervision. All the important parameters of the software can be easily specified by the user.

The system was implemented on a dual Pentium PC with a Matrox C80 vision processing board. The software runs in real-time at a video frame rate of 15 frames/second. The detection accuracy of the system is close to 90% and the classification accuracy is around 70%. Experimental results from highway scenes are provided which demonstrate the effectiveness of the method.

In addition to the proposed system, we also evaluated other non-vision based sensors for classifying vehicles. The *AUTOSENSE II* sensor which is a laser range-finder was tested by us. This report provides a description of the sensor, the results, and a discussion of the advantages and limitations of the *AUTOSENSE II*.

TABLE OF CONTENTS

INTRODUCTION	1
OVERVIEW	1
RELATED WORK	2
VEHICLE DETECTION	5
OVERVIEW	5
MOTION SEGMENTATION	6
BLOB TRACKING	6
RECOVERY OF VEHICLE PARAMETERS	7
VEHICLE IDENTIFICATION	8
VEHICLE TRACKING	9
CLASSIFICATION	10
RESULTS	13
LIMITATIONS	19
ALTERNATIVE SENSORS	21
INTRODUCTION	21
EXPERIMENTAL SETUP	23
COMPARISON WITH GROUND TRUTH	26
ADVANTAGES OF THE <i>AUTOSENSE II</i> SENSOR	26
LIMITATIONS OF THE <i>AUTOSENSE II</i> SENSOR	27
CONCLUSIONS AND FUTURE WORK	29
REFERENCES	31
APPENDIX A	33
INSTRUCTIONS FOR USING THE SOFTWARE	33
APPENDIX B	35
RESULTS FROM THE <i>AUTOSENSE II</i> SENSOR	35
APPENDIX C	43
SOFTWARE LISTINGS	43

LIST OF FIGURES

Figure 1	Computation of the vehicle distance from the camera	11
Figure 2	Calculation of the vehicle length	12
Figure 3	Frame 0 of image sequence	14
Figure 4	Frame 2 of image sequence	14
Figure 5	Frame 0 edge detected	14
Figure 6	Frame 2 edge detected	14
Figure 7	XORing of the two images	15
Figure 8	After performing 3 dilations	15
Figure 9	Identification and classification of the vehicle	15
Figure 10	Detection of vehicles	16
Figure 11	Correct classification of a truck	16
Figure 12	More classification examples	17
Figure 13	Blobs 5 and 10 merge to form blob 14	17
Figure 14	However they are still tracked as one vehicle	17
Figure 15	Blob 83 splits into two blobs – blobs 83 and 87	18
Figure 16	These two blobs are clustered together to form one vehicle 15	18
Figure 17	Blob 1 splits into blobs 3 and 4	18
Figure 18	This is detected while tracking vehicle 0	18
Figure 19	The <i>AUTOSENSE II</i> sensor	21
Figure 20	Schematic diagram of the operation of the <i>AUTOSENSE II</i>	22
Figure 21	A view of Pleasant St	23
Figure 22	The sensor mounted on an overhead bridge above Pleasant St	24
Figure 23	A view of the <i>AUTOSENSE II</i> sensor as a bus passes underneath it	24
Figure 24	A view of Washington Ave	25
Figure 25	The <i>AUTOSENSE II</i> sensor mounted above Washington Ave	25
Figure 26	Image of traffic as seen by the <i>AUTOSENSE II</i> sensor	26

CHAPTER 1

INTRODUCTION

OVERVIEW

Traffic management and information systems rely on a suite of sensors for estimating traffic parameters. Currently, magnetic loop detectors are often used to count vehicles passing over them. Vision-based video monitoring systems offer a number of advantages. In addition to vehicle counts, a much larger set of traffic parameters such as vehicle classifications, lane changes, etc. can be measured. Besides, cameras are much less disruptive to install than loop detectors.

Vehicle classification is important in the computation of the percentages of vehicle classes that use state-aid streets and highways. The current situation is described by outdated data and often, human operators manually count vehicles at a specific street. The use of an automated system can lead to accurate design of pavements (e.g., the decision about thickness) with obvious results in cost and quality. Even in metro areas, there is a need for data about vehicle classes that use a particular street. A classification system like the one proposed here can provide important data for a particular design scenario.

Our system uses a single camera mounted on a pole or other tall structure, looking down on the traffic scene. It can be used for detecting and classifying vehicles in multiple lanes. Besides the camera parameters (focal length, height, pan angle and tilt angle) and direction of traffic, it requires no other initialization.

The report starts by describing an overview of related work, then a description of our approach is included, experimental results are presented, and finally conclusions are drawn.

RELATED WORK

Tracking moving vehicles in video streams has been an active area of research in computer vision. In [1] a real time system for measuring traffic parameters is described. It uses a feature-based method along with occlusion reasoning for tracking vehicles in congested traffic scenes. In order to handle occlusions, instead of tracking entire vehicles, vehicle sub-features are tracked. This approach however is very computationally expensive. In [4] a moving object recognition method is described that uses an adaptive background subtraction technique to separate vehicles from the background. The background is modeled as a slow time-varying image sequence, which allows it to adapt to changes in lighting and weather conditions. In a related work described in [8] pedestrians are tracked and counted using a single camera. The images from the input image sequence are segmented using background subtraction. The resulting connected regions (blobs) are then grouped together into pedestrians and tracked. Merging and splitting of blobs is treated as a graph optimization problem. In [9] a system for detecting lane changes of vehicles in a traffic scene is introduced. The approach is similar to the one described in [8] with the addition that trajectories of the vehicles are determined to detect lane changes.

Despite the large amount of literature on vehicle detection and tracking, there has been very little work done in the field of vehicle classification. This is because vehicle classification is an inherently hard problem. Moreover, detection and tracking are simply preliminary steps in the task of vehicle classification. Given the wide variety of shapes and sizes of vehicles within a single category alone, it is difficult to categorize vehicles using simple parameters. This task is made even more difficult when multiple categories are desired. In real-world traffic scenes, occlusions, shadows, camera noise, changes in lighting and weather conditions, etc. are a fact of life. In addition, stereo cameras are rarely used for traffic monitoring. This makes the recovery of vehicle parameters – such as length, width, height etc, even more difficult given a single camera

view. The inherent complexity of stereo algorithms and the need to solve the correspondence problem makes them unfeasible for real-time applications. In [7] a vehicle tracking and classification system is described that can categorize moving objects as vehicles or humans. However, it does not further classify the vehicles into various classes. In [5] a object classification approach that uses parameterized 3D-models is described. The system uses a 3D polyhedral model to classify vehicles in a traffic sequence. The system uses a generic vehicle model based on the shape of a typical sedan. The underlying assumption being that in typical traffic scenes, cars are more common than trucks or other types of vehicles. To be useful, any classification system should categorize vehicles into a sufficiently large number of classes, however as the number of categories increases, the processing time needed also rises. Therefore, a hierarchical classification method is needed which can quickly categorize vehicles at a coarse granularity. Then depending on the application, further classification at the desired level of granularity should be done.

CHAPTER 2

VEHICLE DETECTION

OVERVIEW

The system proposed here consists of six stages:

1. Motion Segmentation: In this stage, regions of motion are identified and extracted using a temporal differencing approach.
2. Blob Tracking: The result of the motion segmentation step is a collection of connected regions (blobs). The blob tracking stage tracks blobs over a sequence of images using a spatial matching method.
3. Recovery of Vehicle Parameters: To enable accurate classification of the vehicles, the vehicle parameters such as length, width, and height need to be recovered from the 2D projections of the vehicles. This stage uses information about the camera's location and makes use of the fact that in a traffic scene, all motion is along the ground plane.
4. Vehicle Identification: Our system assumes that a vehicle may be made up of multiple blobs. This stage groups the tracked blobs from the previous stage into vehicles. At this stage, the vehicles formed are just hypotheses. The hypotheses can be refined later using information from the other stages.
5. Vehicle Tracking: For robust and accurate detection of vehicles, our system does tracking at two levels – blob level, and the vehicle level. At the vehicle level, tracking is done using Kalman filtering.
6. Vehicle Classification: After vehicles have been detected and tracked, they are classified into various categories.

The following sections describe each of these stages in more detail.

MOTION SEGMENTATION

The first step in detecting objects is segmenting the image to separate the vehicles from the background. There are various approaches to this, with varying degrees of effectiveness. To be useful, the segmentation method needs to accurately separate vehicles from the background, be fast enough to operate in real time, be insensitive to lighting and weather conditions, and require a minimal amount of supplementary information. In [4], a segmentation approach using adaptive background subtraction is described. Though this method has the advantage that it adapts to changes in lighting and weather conditions, it needs to be initialized with an image of the background without any vehicles present. Another approach is time differencing, (used in [7]) which consists of subtracting consequent frames (or frames a fixed number apart). This method too is insensitive to lighting conditions and has the further advantage of not requiring initialization with a background image. However, this method produces many small blobs that are difficult to separate from noise.

Our approach is similar to the time-differencing approach. However instead of simply subtracting consequent frames, it performs an edge detection on two consecutive frames. The two edge-detected images are then combined using a logical XOR operation. This produces a clear outline of only the vehicle; and the background (since it is static) is removed (Figures 3 – 7). After applying a size filter to remove noise and performing a couple of dilation steps, blobs are produced (Figure 8).

BLOB TRACKING

The blob tracking stage relates blobs in frame i to blobs in frame $i+1$. This is done using a spatial locality constraint matching. A blob in frame $i + 1$ will be spatially close to its location in frame i .

To relate a blob in the current frame to one in the previous frame, its location is compared to the locations of blobs in the previous frame. For each blob in the current frame, a blob with the minimum distance (below a threshold) and whose size is similar is searched for, in the previous frame. A new blob is initialized when no blob in the previous frame matches a blob in the current frame. To handle momentary disappearance of blobs, blobs are tracked even if they are not present in the current frame. Each time a blob in the previous frame is not matched to a blob in the current frame, its “age” is incremented. Blobs whose “age” increases above a threshold are removed. To remove noise that was not filtered by the size filter, blobs that do not show significant motion are removed. Blobs can split or merge with other blobs. Instead of explicitly handling splitting and merging at the blob level, this burden is passed onto the vehicle level to handle.

RECOVERY OF VEHICLE PARAMETERS

To be able to detect and classify vehicles, the location, length, width and velocity of the blobs (which are vehicle fragments) needs to be recovered from the image. To enable this recovery, the input image is transformed using translations and affine rotations so that motion of the vehicles is only along one axis. This is a reasonable restriction, since in a traffic sequence, motion occurs only along the ground plane. In the test data we used, the image is rotated so that all motion is along the x -axis only. Using this knowledge and information about the camera parameters, we can extract the distance of the blobs from the camera. The distance is calculated as shown in Figure 1.

The perspective equation (from Figure 1) gives us:

$$y' = f \cdot \frac{Y}{Z} \tag{1}$$

$$y' = f \cdot \tan \delta \tag{2}$$

From Figure 1, it can be seen that

$$\delta = \gamma - \alpha \quad (3)$$

and

$$\gamma = \tan^{-1} \frac{h}{Z_w} \quad (4)$$

therefore,

$$y' = f \cdot \tan \left(\left(\tan^{-1} \frac{h}{Z_w} \right) - \alpha \right) \quad (5)$$

From the Equation (5), the distance to the vehicle (Z_w) can be calculated. To calculate the length of the vehicle, we do the following steps (refer to Figure 2). x_1' and x_2' are the image coordinates of X_1 and X_2 respectively. The length of the vehicle is $|X_1 - X_2|$. From Figure 2,

$$Z_r = \frac{Z_w}{\cos \beta} \quad (6)$$

$$x_1' = f \cdot \frac{X_1}{Z_r} \quad (7)$$

$$= f \cdot \frac{X_1}{Z_w} \cos \beta \quad (8)$$

$$X_1 = \frac{x_1' \cdot Z_w}{f \cdot \cos \beta} \quad (9)$$

where Z_w is as calculated from Equation (5) above.

VEHICLE IDENTIFICATION

A vehicle is made up of blobs. A vehicle in the image may appear as multiple blobs. The vehicle identification stage groups blobs together to form vehicles. New blobs that do not belong to any vehicle are called *orphan blobs*. A vehicle is modeled as a rectangular patch whose dimensions

depend on the dimensions of its constituent blobs. Thresholds are set for the minimum and maximum sizes of vehicles based on typical dimensions of vehicles. A new vehicle is created when an orphan blob is created which is of sufficient size, or a sufficient number of orphan blobs that have similar characteristics (spatial proximity and velocity) can be clustered together to form a vehicle.

VEHICLE TRACKING

Our vehicle model is based on the assumption that the scene has a flat ground. A vehicle is modeled as a rectangular patch whose dimensions depend on its location in the image. The dimensions are equal to the projection of the vehicle at the corresponding location in the scene. The patch is assumed to move with a constant velocity in the scene coordinate system.

The following describes one tracking cycle. More details and the system equations can be found in [8].

1. Relating vehicles to blobs

The relationship between blobs and vehicles is determined as explained above in the Vehicle Identification section.

2. Prediction

Kalman filtering is used to predict the position of the vehicle in the subsequent frame. The velocity of the vehicle is calculated from the velocity of its blobs. Using the vehicle velocity, the position of the vehicle in the current frame, and the time elapsed since the last frame, the position of the vehicle in the current frame is predicted.

3. Calculating vehicle positions.

We use a heuristic in which each vehicle patch is moved around its current location to cover as much as possible of the blobs related to this vehicle. This is taken to be the actual location of the vehicle.

4. Estimation

A measurement is a location in the image coordinate system as computed in the previous subsection. The prediction parameters are updated to reduce the error between the predicted and measured positions of the vehicle.

Since splitting and merging of blobs is not handled at the blob level, it has to be taken into account at the vehicle level. During each frame, when a vehicle is updated, its new dimensions (length and height) are compared with its dimensions in the previous frame. If the new dimensions differ by more than a fixed amount (50% in our experiments), it implies that some of the constituent blobs of this vehicle have either split or merged with other blobs. A decrease in length implies splitting of blobs, whereas an increase indicates merging of blobs. A split implies that a new blob has been created in the current frame that has not been assigned to any vehicle, i.e. an orphan blob. When a decrease in length of a vehicle is detected, the system searches within the set of orphan blobs for blobs that can be clustered with the blobs of this vehicle. The criteria used for clustering is spatial proximity, similar velocity and the sum of the lengths (and heights) of the orphan blobs and existing blobs should not exceed the maximum length (height) threshold. Merging does not need to be explicitly handled. The blobs that have merged are simply replaced with the merged blob. The earlier blobs will be removed for old age during blob tracking.

CLASSIFICATION

The final goal of our system is to be able to do a vehicle classification at multiple levels of granularity. Currently we are classifying vehicles into two categories (based on the needs of the funding agency):

1. Trucks
2. Other vehicles

This classification is made based on the dimensions of the vehicles. Since we calculate the actual length and height of the vehicles, the category of a vehicle can be determined based on its length

and height. Based on typical values, vehicles having length greater than 550 cm. and height greater than 400 cm are considered trucks, while all other vehicles are classified as non-trucks.

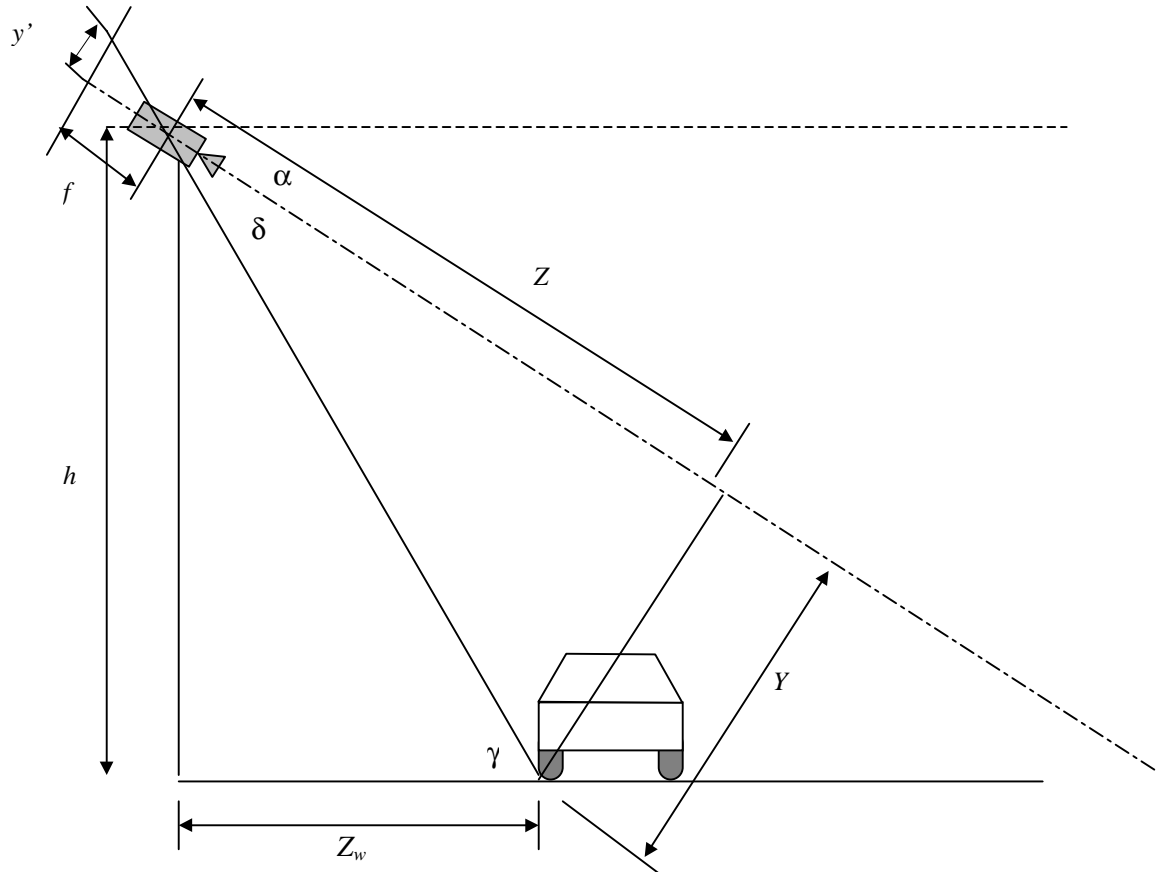


Figure 1 :Computation of the vehicle distance from the camera (α is the camera tilt angle, h is the height of the camera, and Z_w is the distance of the object from the camera, f is the focal length of the camera, y' is the y -coordinate of the point, and Z is the distance to the point along the optical axis).

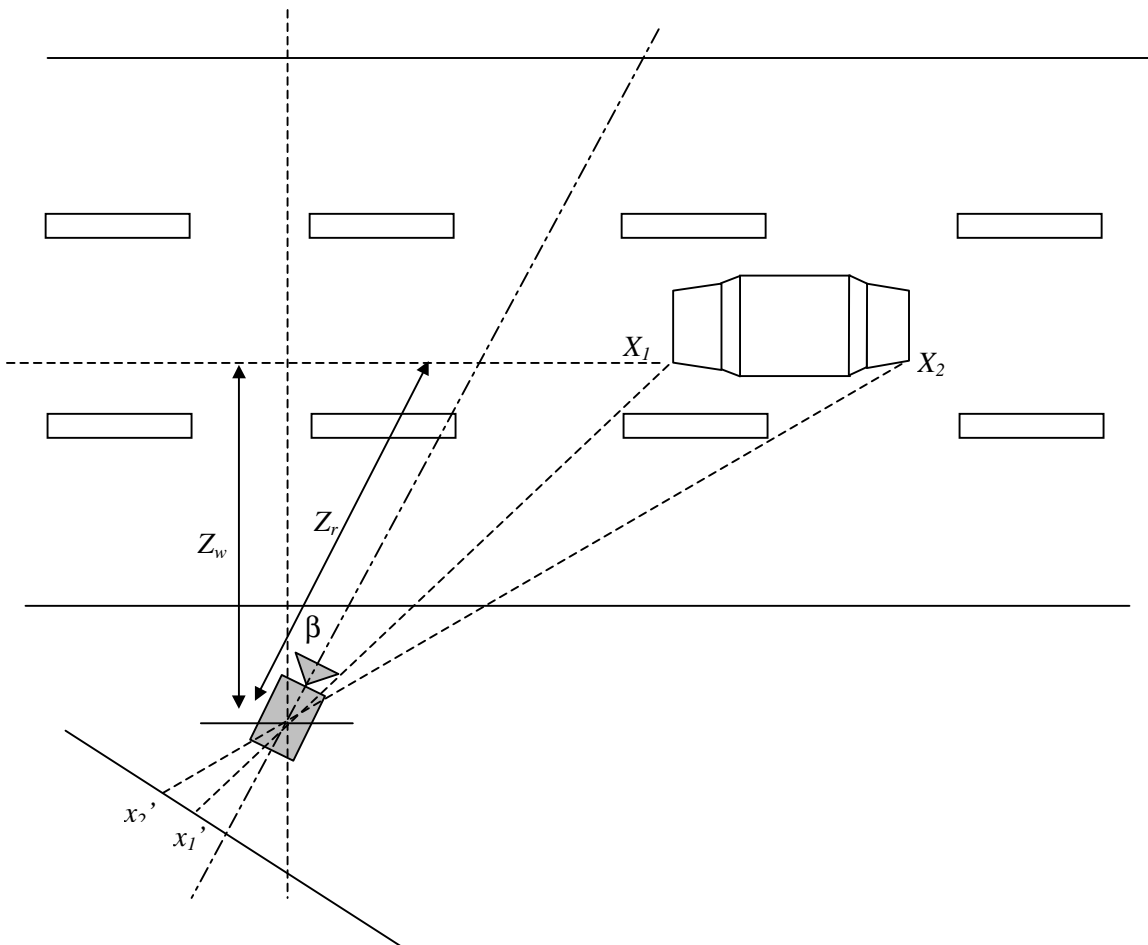


Figure 2 : Calculation of the vehicle length. β is the pan angle of the camera, Z_w is the vertical distance to the vehicle, and Z_r is the distance to the vehicle from the camera (along the optical axis).

CHAPTER 3

RESULTS

The system was implemented on a dual Pentium 400 MHz PC equipped with a C80 Matrox Genesis vision board. We tested the system on image sequences of highway scenes. The system is able to track and classify most vehicles successfully. We were able to achieve a correct classification rate of 70%, and a frame rate of 15 fps. Figures 9 – 18 show the results of our system. With more optimized algorithms, the processing time per frame can be reduced significantly.

There have been cases where the system is unable to do the classification correctly. When multiple vehicles move together, with approximately the same velocity, they tend to get grouped together as one vehicle. Also, the presence of shadows can cause the system to classify vehicles incorrectly. We are currently considering several remedies to handle these situations.



Figure 3 : Frame 0 of the input image sequence.



Figure 4 : Frame 2.

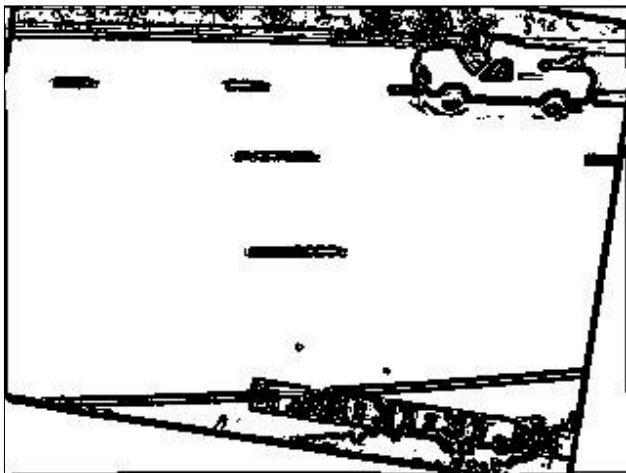


Figure 5 : Frame 0 edge detected.

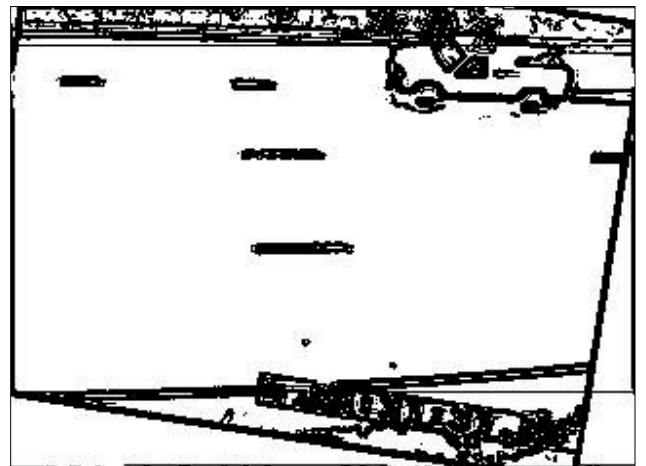


Figure 6 : Frame 2 edge detected.

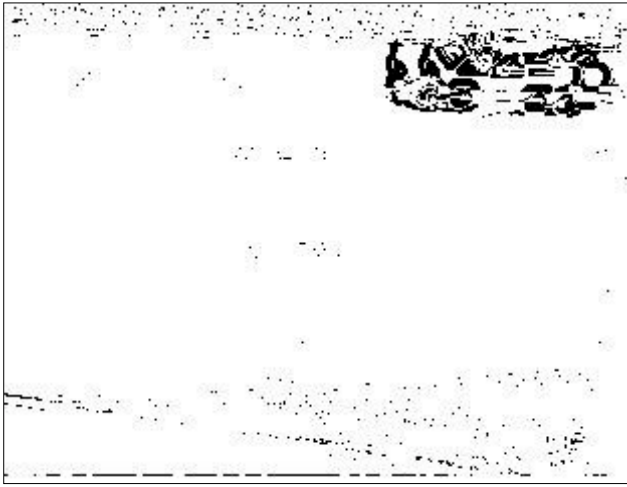


Figure 7 : XORing images from Figs. 5 and 6 .

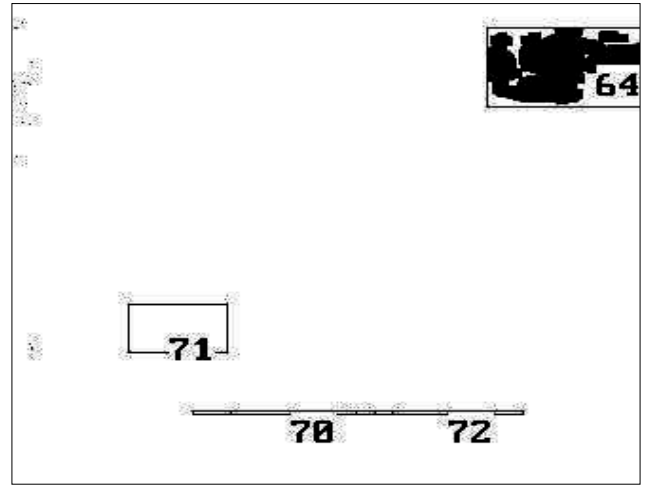


Figure 8 After performing 3 dilations.



Figure 9 : Identification and classification of the vehicle.

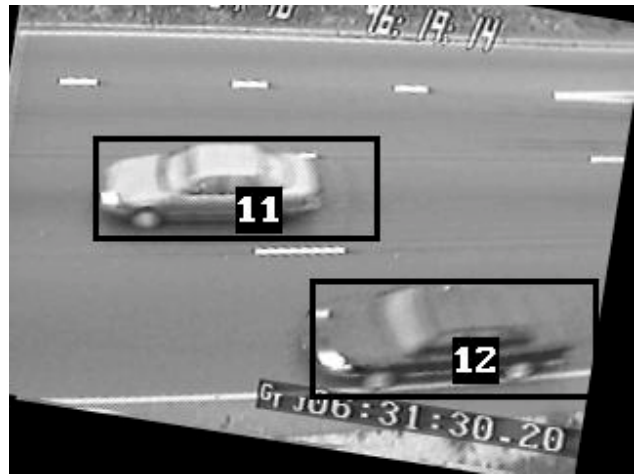
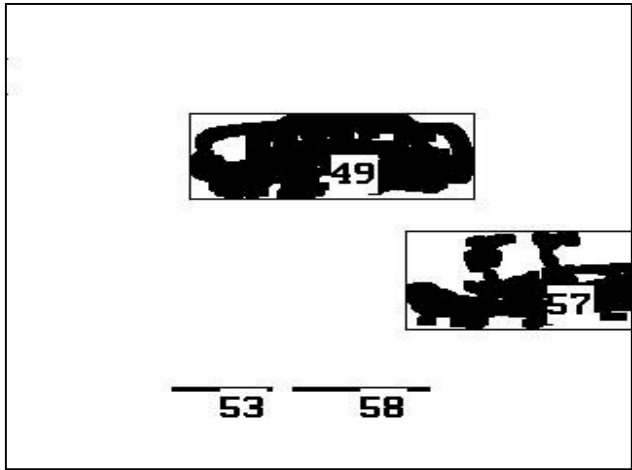


Figure 10 : Detection of vehicles.

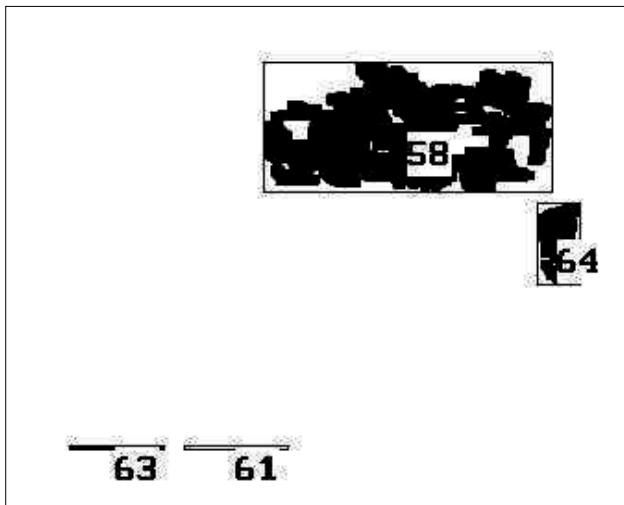


Figure 11 : Correct classification of a truck.

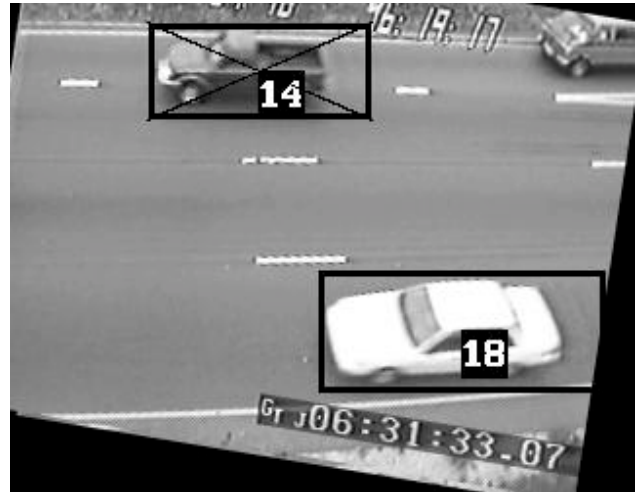
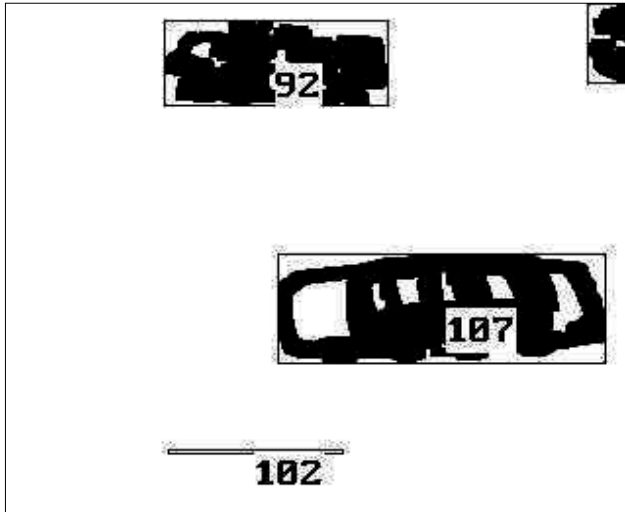


Figure 12 : More classification examples.

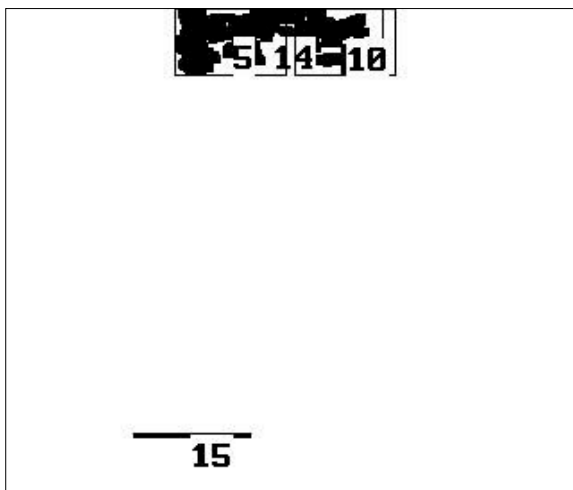


Figure 13 : Blobs 5 and 10 merge to form blob 14.

Figure 14 : However, they are still tracked as one vehicle.

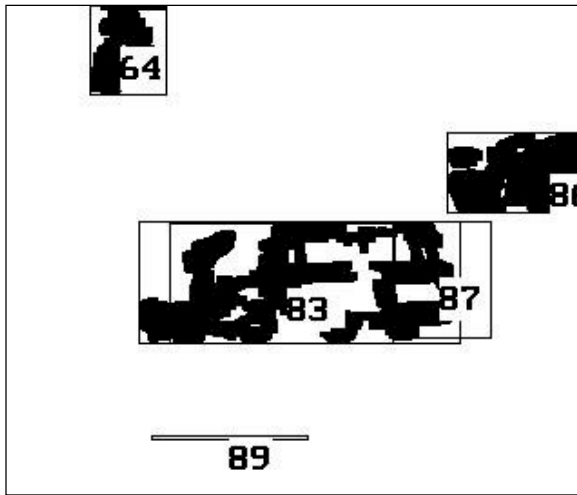


Figure 15 : Blob 83 splits into two blobs – blobs 83 and 87.

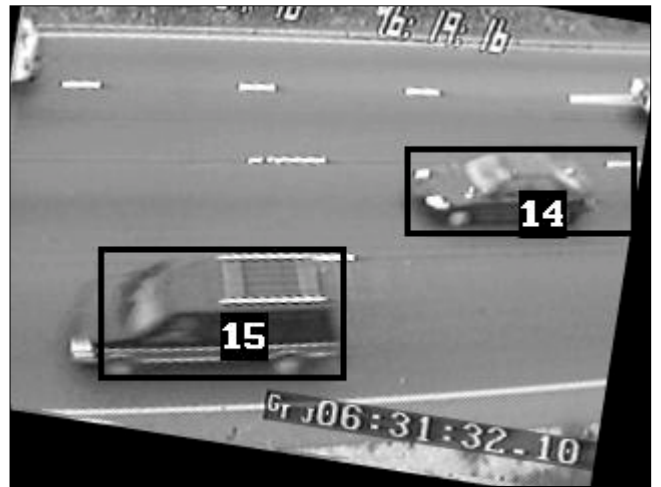


Figure 16 : These two blobs are clustered together to form one vehicle – vehicle 15.

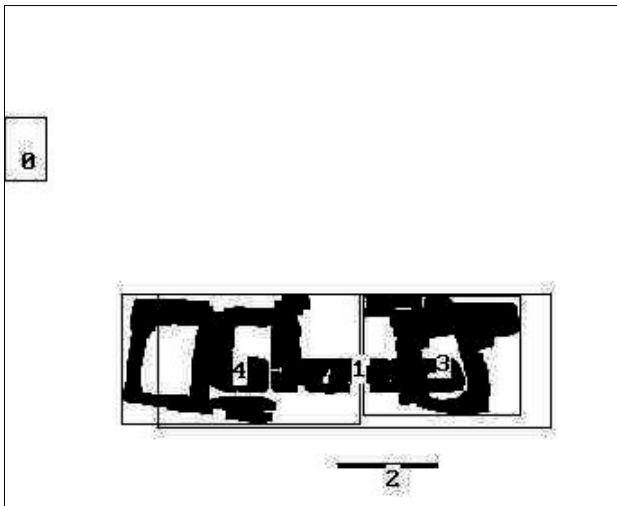


Figure 17 : Blob 1 splits into blobs 3 and 4.

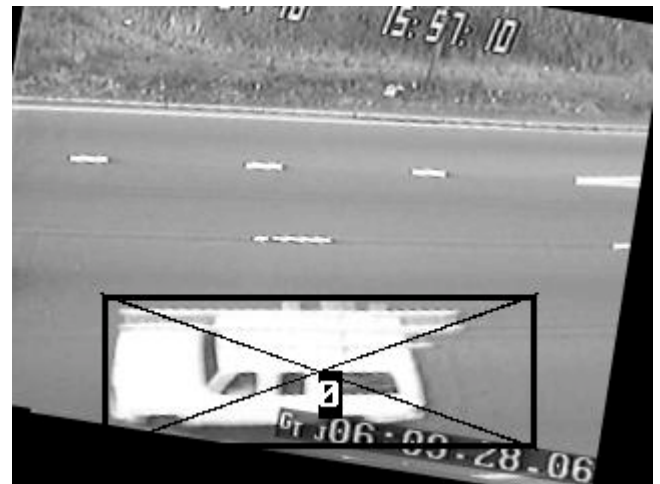


Figure 18 : This is detected while tracking vehicle 0, and these two blobs are clustered and form vehicle 0

CHAPTER 4

LIMITATIONS

These are the limitations of the vehicle detection and classification algorithms that we have described so far:

- The program can only detect and classify vehicles moving in a single direction. This is a limitation of the software. The algorithms will have to be modified to analyze multi-directional traffic.
- The algorithms assume that there is significant motion in the scene between successive frames of the video sequence. If this assumption is not valid for a particular traffic scene, then the accuracy of the results produced will be affected.
- The program cannot reliably analyze scenes that have strong shadows present in them.
- The software cannot work correctly in very low-light conditions.
- In scenes where the density of traffic is very high, causing many vehicles to occlude each other, the algorithms could detect multiple vehicles as a single vehicle, thus affecting the count and also causing a misclassification.

CHAPTER 5

ALTERNATIVE SENSORS

INTRODUCTION

We looked at other methods of doing vehicle classification using sensors other than CCD cameras. Specifically, we looked at the Autosense II sensor from Schwarz Electro-Optics Inc. This is an invisible-beam laser range-finder that does overhead imaging of vehicles to provide size and classification measurements.

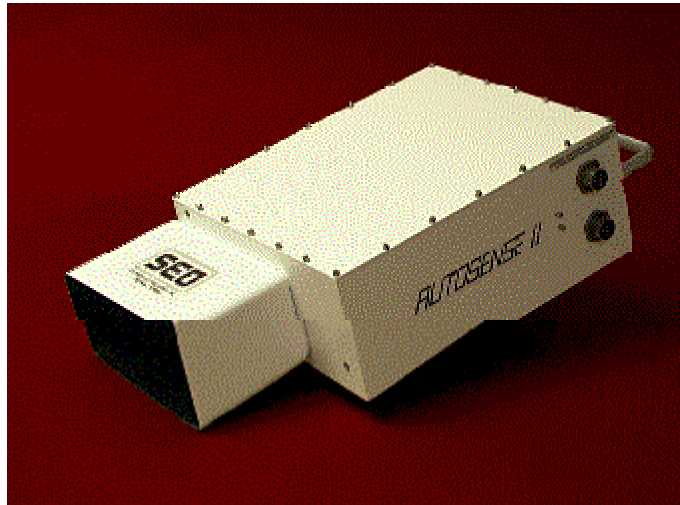


Figure 19 : The AUTONSENSE II sensor.

The *AUTONSENSE II* is mounted above the road at a height of at least 23 feet. Two laser beams scan the roadway by taking 30 range measurements across the width of the road at two locations beneath the sensor. Each set of 30 range measurements forms a line across the road with a 10 degree separation between lines. At a mounting height of 23 feet, a 10 degree separation equals 4 feet between lines. When a vehicle enters the beam, the measured distance decreases and the corresponding vehicle height is calculated using simple geometry and time of flight measurements. As the vehicle progresses, the second beam is also broken in the same manner. The *AUTONSENSE II* calculates the time it takes a vehicle to break both beams, using the beam separation distance, the speed of the vehicle is also calculated. Consecutive range samples are analyzed to generate a profile of the vehicle in view. This vehicle profile is then processed by the sensor to classify the vehicle into 13 different categories.

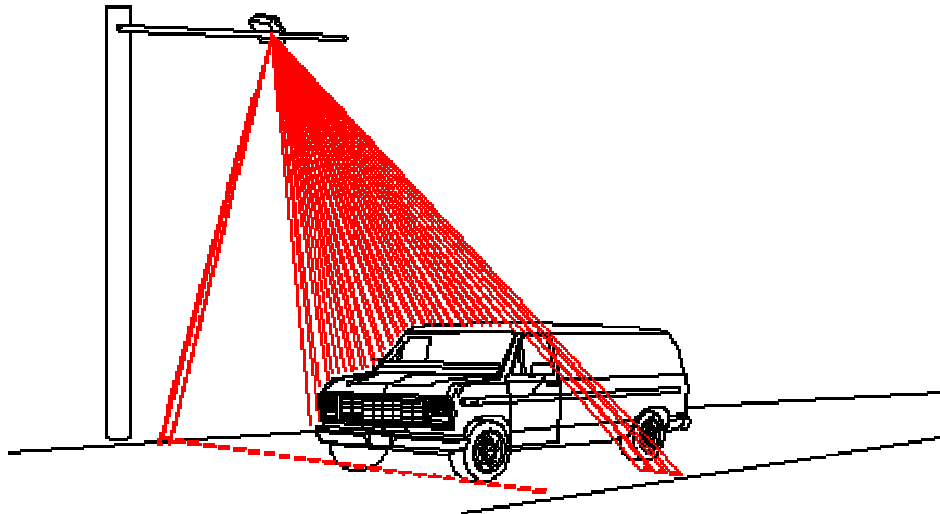


Figure 20 : Schematic diagram of the operation of the AUTOSENSE II.

The *AUTOSENSE II* transmits 5 messages for each vehicle that is detected within its field of view. The messages and the order in which it is transmitted are listed:

- #1 First Beam Vehicle Detection Message
- #2 Second Beam Vehicle Detection Message
- #3 First Beam End of Vehicle Message
- #4 Second Beam End of Vehicle Message
- #5 Vehicle Classification Message

The first four messages uniquely identify each vehicle and its position in the lane. The classification message includes vehicle classification, classification confidence percentage, height, length, width and speed.

The *AUTOSENSE II* can classify vehicles into the following five categories:

1. Car
2. Pickup/Van/SUV
3. Bus
4. Tractor
5. Motorcycle

Besides these five basic categories, the *AUTOSENSE II* can also detect the presence or absence of a trailer and hence can provide eight additional sub-categories.

EXPERIMENTAL SETUP

We tested the *AUTOSENSE II* for 3 lane-hours in various weather and lighting conditions. The sensor was tested at two different locations - Washington Ave. and Pleasant St. Washington Ave. is a three lane street, with vehicles usually by at speeds of around 50-60 mph. Pleasant St. is a single lane road and traffic on it merges with Washington Ave. The average speed of vehicles on Pleasant St. is approximately 20-30 mph.



Figure 21 : A view of Pleasant Street.



Figure 22 : The sensor mounted on an overhead bridge above Pleasant Street.



**Figure 23: A view of the AUTOSENSE II sensor as a bus passes underneath it
(the sensor is shown highlighted with a white box around it).**



Figure 24 : A view of Washington Avenue.



**Figure 25 : The *AUTONSENSE II* sensor mounted above Washington Ave.
(sensor is highlighted with a black box around it).**



Figure 26 : Image of traffic as seen by the *AUTOSENSE II* sensor as a car passes underneath it.

RESULTS

The results from the *AUTOSENSE II* sensor are given in Appendix B. These results have been post-processed using Microsoft Excel. The sensor does not provide results in the format shown.

COMPARISON WITH GROUND TRUTH

The results of the *AUTOSENSE II* sensor were compared to manually collected data. These comparisons indicate that the detection accuracy of the *AUTOSENSE II* sensor is approximately 99%. The only cases it failed to detect a vehicle correctly were when the vehicle was not entirely within the lane that the sensor was centered on. This would sometimes lead the sensor to not detect the vehicle or misclassify it. The classification accuracy, too was around 99%. The cases where it failed to classify a vehicle correctly were usually cases where the vehicle was a SUV whose length was smaller than that of average SUVs (for e.g. a Honda CR-V). In most other cases, the sensor did classify the vehicles correctly.

ADVANTAGES OF THE *AUTOSENSE II* SENSOR

After testing the sensor for a significant amount of time in various and adverse conditions we have discovered that these are the advantages of the *AUTOSENSE II* sensor.

- Very high detection and classification accuracy

- Not affected by lighting and/or weather conditions. The sensor can be used even under zero-light conditions.

LIMITATIONS OF THE *AUTOSENSE II* SENSOR

Though the *AUTOSENSE II* sensor has very high detection and classification accuracy, in our opinion, it has some limitations as detailed below.

- The *AUTOSENSE II* sensor can only detect and classify vehicles in a single lane. However, this limitation can be overcome by using the newly introduced *AUTOSENSE III* sensor, which can analyse data in multiple lanes.
- The sensor has very rigid mounting requirements which could make it unsuitable for general purpose use in any situation. Specifically, it requires overhead mounting, at a height of at least 23 feet. The sensor has to be mounted vertically, and the angle it makes with the vertical can be at most 5 degrees. Any obstructions in the path of the beam will cause the sensor to provide erroneous results. The sensor is more suited for a permanent installation, and is not amenable to temporary collection of data at some site.
- The sensor requires line voltage (110V AC) and has to be connected to a computer via a serial cable. Due to limitations of the serial protocol, there are limits on the length of the cable that can be used (a maximum of 40 feet) and hence on the distance that the computer can be away from the sensor.
- The sensor has to be connected to an on-site computer. It is not possible to simply collect the data from the sensor and then process it offline (as for example can be done with cameras and video tape). Hence, there is the additional cost of installing a computer on-site.
- Since the data collection, analysis and classification is done by proprietary software provided by the manufacturer, it is not possible to do a finer or coarser classification or change the categorization of vehicles as provided by the sensor.
- The sensor cannot analyze data from multi-directional traffic. To analyze such data would require the use of multiple sensors, one for each lane.
- The sensor can only analyze data from scenes where the traffic is moving perpendicular to the direction of the laser beams. The sensor cannot be used in a scene, where say the vehicles are turning.

- Though the sensor provides accurate data for the count of vehicles, speed and classification, it cannot be used to provide other data, which a video camera can provide, for example, tracking information.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

We have presented a model-based vehicle tracking and classification system capable of working robustly under most circumstances. The system is general enough to be capable of detecting, tracking and classifying vehicles without requiring any scene-specific knowledge or manual initialization. In addition to the vehicle category, the system provides location and velocity information for each vehicle as long as it is visible. Initial experimental results from highway scenes were presented.

To enable classification into a larger number of categories, we intend to use a non-rigid model-based approach to classify vehicles. Parameterized 3D models of exemplars of each category will be used. Given the camera location, tilt and pan angles, a 2D projection of the model will be formed from this viewpoint. This projection will be compared with the vehicles in the image to determine the class of the vehicle.

REFERENCES

1. D. Beymer, P. McLauchlan, B. Coifman and J. Malik, "A Real-Time Computer Vision System for Measuring Traffic Parameters," *IEEE Conf. Computer Vision and Pattern Recognition*, June 1997, Puerto Rico.
2. M. Burden and M. Bell, "Vehicle classification using stereo vision," in *Proc. of the Sixth International Conference on Image Processing and its Applications*, 1997.
3. A. De La Escalera, L.E. Moreno, M.A. Salichs, and J.M. Armingol, "Road traffic sign detection and classification," *IEEE Transactions on Industrial Electronics*, December 1997.
4. Klaus-Peter Karmann and Achim von Brandt, "Moving Object Recognition Using an Adaptive Background Memory," in *Time-Varying Image Processing and Moving Object Recognition, 2* – edited by V. Capellini, 1990.
5. D. Koller, "Moving Object Recognition and Classification based on Recursive Shape Parameter Estimation," *12th Israel Conference on Artificial Intelligence, Computer Vision*, December 27-28, 1993.
6. D. Koller, J. Weber, T. Huang, G. Osawara, B. Rao and S. Russel, "Towards Robust Automatic Traffic Scene Analysis in Real-Time," in *Proc. of the 12th Int'l Conference on Pattern Recognition* 1994.
7. Alan J. Lipton, Hironobu Fujiyoshi and Raju S. Patil, "Moving Target Classification and Tracking from Real-Time Video," in *Proc. of the Image Understanding Workshop*, 1998.
8. Osama Masoud and Nikolaos Papanikolopoulos, "Robust Pedestrian Tracking Using a Model-Based Approach," in *Proc. IEEE Conference on Intelligent Transportation Systems*, pp. 338-343, November 1997.
9. Osama Masoud and Nikolaos Papanikolopoulos, "Vision Based Monitoring of Weaving Sections," in *Proc. IEEE Conference on Intelligent Transportation Systems*, October 1999.

10. S. Meller, N. Zabaronik, I. Ghoreishian, J. Allison, V. Arya, M. de Vries, and R. Claus, "Performance of fiber optic vehicle sensors for highway axle detection," in *Proc. of SPIE (Int. Soc. Opt. Eng.)*, Vol. 2902, 1997.
11. W. Schwartz and R. Olson, "Wide-area traffic-surveillance (WATS) system," in *Proc. of SPIE (Int. Soc. Opt. Eng.)*, Vol. 2902, 1997.
12. H. Tien, B. Lau, and Y. Park, "Vehicle detection and classification in shadowy traffic images using wavelets and neural networks," in *Proc. of SPIE (Int. Soc. Opt. Eng.)*, Vol. 2902, 1997.

APPENDIX A INSTRUCTIONS FOR USING THE SOFTWARE

The software is completely self-running requiring no user-interaction. However, it has enough flexibility to allow user-configurability. All configuration by the user is done by means of a parameter file. This is a plain-text file which can be edited by the user. A very basic structure is imposed on the format of this file. This format has been kept simple enough for most users to be able to change the parameters easily. Each line of the file corresponds to one parameter. Each line consists of a name – value pair. The name and value are separated by a space and colon (:) character. Thus each line in the parameter file looks like:

Name : Value

With at least one space between the name and the colon, and the colon and the value. Names can consist of any character (except space and tab). The following parameters are configurable by the user through this file:

1. Camera_Height The height of the camera from the ground (in centimeters).
2. Camera_Distance Horizontal distance of the camera from the nearest lane to it (in cm).
3. Focal_Length The focal length of the camera (in cm).
4. Tilt_Angle The tilt angle of the camera in degrees, measured counterclockwise around the horizontal axis.
5. Pan_Angle The pan angle of the camera in degrees measured counterclockwise around the vertical axis.
6. Resolution The resolution of the camera in pixels/cm.
7. Image_Width The width of the image in pixels.
8. Image_Height The height of the image in pixels.
9. Number_Lanes The number of lanes in the scene to be analyzed.
10. Interval Time interval at which to generate the records (in seconds).
11. Output_File The name of the file in which the output is to be recorded.

These parameters can be specified in any order, but they must be spelt exactly as shown here. In addition, comment lines can be inserted in the file by entering the # character as the first character on a line. Everything on that line will be considered a comment and ignored by the program.

In addition, all the parameters also have default values. Parameters that are not specified in the parameter file are assigned the default values. The default values for the parameters are:

1. Camera_Height 977.35
2. Camera_Distance 686
3. Focal_Length 0.689
4. Tilt_Angle -39.54
5. Pan_Angle -15.0
6. Resolution 1000
7. Image_Width 320
8. Image_Height 240
9. Number_Lanes 4
10. Interval 120
11. Output_File The screen

These values are based on the ones we calculated from the tapes we have been using.

The image width and image height are determined automatically. In most circumstances, these should not be specified via the parameter file.

By default the program looks for a file called “params.ini” in the current working directory. A different file can be specified by giving the file name as the first command line argument to the program. If the program cannot find the file, or there is an error in the syntax of a parameter specification, or the parameter has not been specified in the file, then in any of these circumstances, the program uses the default values for the parameter.

APPENDIX B

RESULTS FROM THE *AUTOSENSE II* SENSOR

Results for Washington Ave. 11/27/99

<hr/>	
15:35:02	
<hr/>	
Motorcycle	0
Car	1
Tractor	0
Bus	0
Pickup/Van/Sport Utility	0
Pickup/Van/Sport Utility w/Trailer	0
Car w/Trailer	0
Bus w/Trailer	0
Average Speed	37
<hr/>	
15:40:00	
<hr/>	
Motorcycle	2
Car	20
Tractor	1
Bus	0
Pickup/Van/Sport Utility	5
Pickup/Van/Sport Utility w/Trailer	0
Car w/Trailer	0
Bus w/Trailer	0
Average Speed	37.8
<hr/>	
15:45:09	
<hr/>	
Motorcycle	0
Car	21
Tractor	0
Bus	1
Pickup/Van/Sport Utility	8
Pickup/Van/Sport Utility w/Trailer	0
Car w/Trailer	0
Bus w/Trailer	0
Average Speed	40

APPENDIX C SOFTWARE LISTINGS

This is the list of files used by the system. The files are appended in the same order as they are listed here.

Header Files

1. BlobClusterer.h
2. Cluster.h
3. Clusterer.h
4. Blob.h
5. BlobCluster.h
6. BlobData.h
7. BlobManager.h
8. BoundingBox.h
9. Camera.h
10. Ini_file_reader.h
11. Parameters.h
12. Reporter.h
13. Vector2d.h
14. Vehicle.h
15. VehicleClassifier.h
16. VisionProcessor.h

Source Files

1. BlobClusterer.cpp
2. Cluster.cpp
3. Clusterer.cpp
4. Blob.cpp
5. BlobCluster.cpp
6. BlobData.cpp
7. BlobManager.cpp
8. BoundingBox.cpp

9. Camera.cpp
10. Ini_file_reader.cpp
11. Parameters.cpp
12. Reporter.cpp
13. Vector2d.cpp
14. Vehicle.cpp
15. VechicleClassifier.cpp
16. VisionProcessor.cpp

```

// BlobCluster.h: interface for the BlobCluster class.
//
////////////////////////////////////

#if
!defined(AFX_BLOBCLUSTER_H__B3ACFCC2_6885_11D3_9175_0040053461F8__
INCLUDED_)
#define
AFX_BLOBCLUSTER_H__B3ACFCC2_6885_11D3_9175_0040053461F8__INCLUD
ED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "cluster.h"
#include "blob.h"
#include "Vector2d.h"
#include <list>
#include <iostream>

using std::ostream;

class BlobCluster : public Cluster
{
public:
    BlobCluster();
    BlobCluster(Blob *blob);
    virtual ~BlobCluster();
    void updateDimensions();
    void removeBlob(Blob *blob);
    void replaceBlobs(BlobCluster *blobs);
    float getLength() { return _box.length(); }
    float getWidth() { return _box.width(); }
    int getNumBlobs() { return _blobs.size(); }
    list<Blob*>& getBlobs() { return _blobs; }
    BoundingBox& getBoundingBox() { return _box; }
    void assignVehicle(Vehicle *veh);

    friend ostream& operator<<(ostream& ostr, BlobCluster &cluster);

private:
    bool _canBeMerged(Cluster &cluster);
    bool _merge(Cluster &cluster);

    double _similarity(Cluster &cluster);

```



```
double _distance(Cluster &cluster);

bool _replacing;
static BlobManager& _blobMgr;
list<Blob*> _blobs;
BoundingBox _box;

Vector2d _imgVelocity;

static const float _MaxClusterLength, _MaxClusterWidth, _MaxBlobClusterDist,
                 _VelocityUpdateFactor;
};

#endif //
!defined(AFX_BLOBCLUSTER_H__B3ACFCC2_6885_11D3_9175_0040053461F8__
INCLUDED_)
```

```

// cluster.h: interface for the Cluster class.
//
////////////////////////////////////

#if
!defined(AFX_CLUSTER_H__B3ACFCC1_6885_11D3_9175_0040053461F8__INCL
UDED_)
#define
AFX_CLUSTER_H__B3ACFCC1_6885_11D3_9175_0040053461F8__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <vector>

#ifdef CLUSTER
#include <iostream>
using std::ostream;
#endif

using std::vector;

class Cluster
{
public:
    Cluster() ;
    virtual ~Cluster() ;

protected:
    float _length;
    float _width;

    virtual bool _merge(Cluster &cluster) = 0;
    virtual double _similarity(Cluster &cluster) = 0;
    virtual double _distance(Cluster &cluster) = 0;
    virtual bool _canBeMerged(Cluster &cluster) = 0;
    virtual float getLength() = 0;
    virtual float getWidth() = 0;

// friend ostream& operator<<(ostream &ostr, Cluster &cluster);
// virtual float center() = 0;

    friend class Clusterer;

```

```
};
```

```
#endif //
```

```
!defined(AFX_CLUSTER_H__B3ACFCC1_6885_11D3_9175_0040053461F8__INCL  
UDED_)
```

```

// Clusterer.h: interface for the Clusterer class.
//
/////////////////////////////////////////////////////////////////

#if
!defined(AFX_CLUSTERER_H__B34C9282_681C_11D3_9175_0040053461F8__INC
LUDED_)
#define
AFX_CLUSTERER_H__B34C9282_681C_11D3_9175_0040053461F8__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <list>
#include <iostream>
#include "cluster.h"

using std::ostream;

class Clusterer
{
public:
    Clusterer();
    virtual ~Clusterer();

    bool expandCluster(Cluster &cluster, std::list<Cluster*> &clusters);
    bool expandCluster(Cluster &cluster, std::list<Cluster*> &clusters,
        std::list<Cluster*>::iterator start);
    void expandClusters(std::list<Cluster*> &cluster, std::list<Cluster*>
&clusterees);

//private:
    class Cl {
    public:
        std::list<Cluster*>::iterator cluster;
        std::list<Cluster*>::iterator clusteree;
        float similarity;

        Cl() { similarity = 0; }
        Cl(float sim, std::list<Cluster*>::iterator iter, std::list<Cluster*>::iterator
iter2) : similarity(sim), cluster(iter), clusteree(iter2) {}
        bool operator> (const Cl& c1) const {
            return similarity > c1.similarity;
        }
    };
};

```

```
        friend ostream& operator<<(ostream&, Cl&);  
};  
  
#endif //  
!defined(AFX_CLUSTERER_H__B34C9282_681C_11D3_9175_0040053461F8__INC  
LUDED_)
```

```

// Blob.h: interface for the Blob class.
//
////////////////////////////////////

#if
!defined(AFX_BLOB_H__FBA75CC2_31B2_11D3_9198_0040053461F8__INCLUDE
D_)
#define
AFX_BLOB_H__FBA75CC2_31B2_11D3_9198_0040053461F8__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <list>
#include "blobData.h"
#include "boundingBox.h"
#include "vector2d.h"

using std::list;

class VisionProcessor;
class Vehicle;

class Blob
{
public:

    Blob(blobData &bdata);
    void update(long x, long y, long minX, long minY, long maxX, long maxY, long
area);
    void update(blobData &bd);
    void show();

    //all the get methods
    Vector2d& getVelocity() {return _velocity; }
    long* getPosition() {return _blobData.centerGravity; }
    BoundingBox& getBoundingBox() {return _boundingBox; }
    long getArea() { return _blobData.area; }
    int getName() { return _name; }
    float distance(Blob* const blob) { return BBox::distance(_boundingBox, blob-
>getBoundingBox()); }
    float seperation(Blob* const blob) { return BBox::seperation(_boundingBox,
blob->getBoundingBox()); }

    void setVehicle(Vehicle *veh) { _vehicle = veh; }

```

```

Vehicle* getVehicle() { return _vehicle; }
int distance(long cg[2]);

private:
    blobData _blobData;
    virtual ~Blob();
    BoundingBox _boundingBox;

    static VisionProcessor *_visProc;
    static int _count;
    int _name;
    Vehicle* _vehicle;
    Vector2d _velocity;

    friend class BlobManager;
    friend class BlobClusterer;

    /* These factors determine how quickly their respective parameters change
    Maybe they shouldn't be constants, but should change depending on certain
    factors. But for now they are statically decided

    const float _PositionUpdateFactor ;
    const float _VelocityUpdateFactor;
    const float _AreaUpdateFactor;
    const float _MinVelocity;
*/

};

#endif //
!defined(AFX_BLOB_H__FBA75CC2_31B2_11D3_9198_0040053461F8__INCLUDE
D_)

```

```

// BlobCluster.h: interface for the BlobCluster class.
//
////////////////////////////////////////////////////////////////

#if
!defined(AFX_BLOBCLUSTER_H__B3ACFCC2_6885_11D3_9175_0040053461F8__
INCLUDED_)
#define
AFX_BLOBCLUSTER_H__B3ACFCC2_6885_11D3_9175_0040053461F8__INCLUD
ED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "cluster.h"
#include "blob.h"
#include "Vector2d.h"
#include <list>
#include <iostream>

using std::ostream;

class BlobCluster : public Cluster
{
public:
    BlobCluster();
    BlobCluster(Blob *blob);
    virtual ~BlobCluster();
    void updateDimensions();
    void removeBlob(Blob *blob);
    void replaceBlobs(BlobCluster *blobs);
    float getLength() { return _box.length(); }
    float getWidth() { return _box.width(); }
    int getNumBlobs() { return _blobs.size(); }
    list<Blob*>& getBlobs() { return _blobs; }
    BoundingBox& getBoundingBox() { return _box; }
    void assignVehicle(Vehicle *veh);

    friend ostream& operator<<(ostream& ostr, BlobCluster &cluster);

private:
    bool _canBeMerged(Cluster &cluster);
    bool _merge(Cluster &cluster);

    double _similarity(Cluster &cluster);

```



```
double _distance(Cluster &cluster);

bool _replacing;
static BlobManager& _blobMgr;
list<Blob*> _blobs;
BoundingBox _box;

Vector2d _imgVelocity;

static const float _MaxClusterLength, _MaxClusterWidth, _MaxBlobClusterDist,
                 _VelocityUpdateFactor;
};

#endif //
!defined(AFX_BLOBCLUSTER_H__B3ACFCC2_6885_11D3_9175_0040053461F8__
INCLUDED_)
```

```

// blobData.h: interface for the blobData class.
//
////////////////////////////////////

#if
!defined(AFX_BLOBDATA_H__3568FFE4_4386_11D3_9159_0040053461F8__INCL
UDED_)
#define
AFX_BLOBDATA_H__3568FFE4_4386_11D3_9159_0040053461F8__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <vector>
#include <iostream>

using std::vector;

typedef struct blobData {
    long label;
    long boundingBox[4];
    long area;
    long centerGravity[2];

    blobData(long lab, long minx, long miny, long maxx, long maxy,
             long ar, long cgx, long cgy) {
        label = lab;
        boundingBox[0] = minx; boundingBox[1] = miny;
        boundingBox[2] = maxx; boundingBox[3] = maxy;
        area = ar;
        centerGravity[0] = cgx; centerGravity[1] = cgy;
    }
} blobData;

#endif //
!defined(AFX_BLOBDATA_H__3568FFE4_4386_11D3_9159_0040053461F8__INCL
UDED_)

```

```

// BlobManager.h: interface for the BlobManager class.
//
////////////////////////////////////

#if
!defined(AFX_BLOBMANAGER_H__CA4842B2_42CF_11D3_919A_0040053461F8_
_INCLUDED_)
#define
AFX_BLOBMANAGER_H__CA4842B2_42CF_11D3_919A_0040053461F8__INCLU
DED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "VisionProcessor.h"
#include "Blob.h"
#include <list>

using std::list;

class BlobManager
{
public:
    virtual ~BlobManager();

    static BlobManager& getInstance();
    void addBlobs(list<blobData*> &lst);
    void removeBlob(Blob* blob);
    void removeBlobs(list<Blob*>& blobs);
    void showBlobs();
    void showMatchedBlobs();
    list<Blob*>& getBlobs() { return _blobs; }

private:
    BlobManager();

    list<Blob*> _blobs;
    static BlobManager * _instance;
    static const unsigned int _MinBlobDistance;
    static const unsigned int _MinBlobDisplacement;
    static const unsigned int _MaxAge;
    static const unsigned int _MaxStaticCount;
    static const float _OverlapThreshold;

};

```

```
#endif //  
!defined(AFX_BLOBMANAGER_H__CA4842B2_42CF_11D3_919A_0040053461F8_  
_INCLUDED_)
```

```

// BoundingBox.h: interface for the BoundingBox class.
//
////////////////////////////////////

#if
!defined(AFX_BOUNDBINGBOX_H__1BABC580_66E4_11D3_9175_0040053461F8__
INCLUDED_)
#define
AFX_BOUNDBINGBOX_H__1BABC580_66E4_11D3_9175_0040053461F8__INCLU
DED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class BoundingBox;

namespace BBox
{
    double seperation(BoundingBox &box1, BoundingBox &box2);
    double distance(BoundingBox &box1, BoundingBox &box2);
    double overlap(BoundingBox &box1, BoundingBox &box2);
};

class BoundingBox
{
public:
    BoundingBox() {}
    BoundingBox(double left, double bottom, double right, double top);
    BoundingBox(float box[4]);
    BoundingBox(double box[4]);
    BoundingBox(long box[4]);
    virtual ~BoundingBox();

    void setCoordinates(float left, float bottom, float right, float top)
    {
        _box[0] = left; _box[1] = bottom;
        _box[2] = right; _box[3] = top;
    }

    void setCoordinates(float box[4])
    {
        _box[0] = box[0]; _box[1] = box[1];
        _box[2] = box[2]; _box[3] = box[3];
    }
}

```

```

void setCoordinates(long box[4])
{
    _box[0] = box[0]; _box[1] = box[1];
    _box[2] = box[2]; _box[3] = box[3];
}

double* coordinates() { return _box; };
void center(double cg[]);
double length() { return (_box[2] - _box[0]); }
double width() { return (_box[3] - _box[1]); }
double seperation(BoundingBox &box1);
double overlap(BoundingBox &box1);
double symOverlap(BoundingBox &box1);
double distance(BoundingBox &box);
void operator+=(BoundingBox &box);
double operator[](int i) { if(i >= 0 && i < 4) return _box[i]; return 0;}

private:
    double _box[4];

    friend double BBox::seperation(BoundingBox &box1, BoundingBox &box2);
    friend double BBox::distance(BoundingBox &box1, BoundingBox &box2);
    friend double BBox::overlap(BoundingBox &box1, BoundingBox &box2);
};

#endif //
!defined(AFX_BOUNDINGBOX_H__1BABC580_66E4_11D3_9175_0040053461F8__
INCLUDED_)

```

```

// BoundingBox.cpp: implementation of the BoundingBox class.
//
////////////////////////////////////

#include "BoundingBox.h"
#include <math.h>
#include <iostream>

using std::cout;
using std::endl;

inline double sqr(double x) { return (x) * (x); }
inline double max(double x1, double x2) { return ( (x1) > (x2) ? (x1) : (x2)); }
inline double min(double x1, double x2) { return ( (x1) < (x2) ? (x1) : (x2)); }

////////////////////////////////////
// Construction/Destruction
////////////////////////////////////

BoundingBox::BoundingBox(double left, double bottom, double right, double top)
{
    _box[0] = left; _box[1] = bottom;
    _box[2] = right; _box[3] = top;
}

BoundingBox::BoundingBox(double box[4])
{
    _box[0] = box[0]; _box[1] = box[1];
    _box[2] = box[2]; _box[3] = box[3];
}

BoundingBox::BoundingBox(long box[4])
{
    _box[0] = box[0]; _box[1] = box[1];
    _box[2] = box[2]; _box[3] = box[3];
}

BoundingBox::~BoundingBox()
{
}

void BoundingBox::operator+= (BoundingBox &bBox)
{
    double *box = bBox.coordinates();
    if(box[0] < _box[0] )

```

```

        _box[0] = box[0];
    if(box[1] < _box[1] )
        _box[1] = box[1];
    if(box[2] > _box[2] )
        _box[2] = box[2];
    if(box[3] > _box[3] )
        _box[3] = box[3];
}

void BoundingBox::center(double cg[])
{
    cg[0] = _box[0] + length()/2;
    cg[1] = _box[1] + width()/2;
}

double BoundingBox::symOverlap(BoundingBox &box1) {
    double ovr1 = overlap(box1);
    double ovr2 = box1.overlap(*this);
    return ovr1 > ovr2 ? ovr1 : ovr2;
}

double BoundingBox::overlap(BoundingBox &box1) {
    //first check if the boxes overlap in x-direction
    double xoverlap = 0, yoverlap = 0;
    double lt, rt, tp, bt;
    if((_box[0] <= box1._box[0]) && (_box[2] >= box1._box[0]))
    {
        lt = box1._box[0];
        rt = min(_box[2], box1._box[2]);
    }
    else
        if((box1._box[0] <= _box[0]) && (box1._box[2] >= _box[0]))
        {
            lt = _box[0];
            rt = min(_box[2], box1._box[2]);
        }
        xoverlap = rt - lt;
#ifdef BOX
    cout << "Left : " << lt << " Right : " << rt << endl;
    cout << "Xoverlap : " << xoverlap << endl;
#endif
}

//check for overlap in y-direction
if((_box[1] <= box1._box[1]) && (_box[3] >= box1._box[1]))
{
    bt = box1._box[1];
    tp = min(_box[3], box1._box[3]);
}

```



```

    }
    else
        if((box1._box[1] <= _box[1]) && (box1._box[3] >= _box[1]))
        {
            bt = _box[1];
            tp = min(_box[3], box1._box[3]);
        }
        yoverlap = tp - bt;

#ifdef BOX
    cout << "Top : " << tp << " Bottom : " << bt << endl;
    cout << "Yoverlap : " << yoverlap << endl;
#endif
    return (xoverlap * yoverlap)/(length()*width());
}

double BoundingBox::distance(BoundingBox &box1)
{
    double cg1[2], cg2[2];
    center(cg1);
    box1.center(cg2);
    double dist = sqrt(sqr(cg1[0] - cg2[0]) + sqr(cg1[1] - cg2[1]));
    return dist;
}

double BBox::distance(BoundingBox &box1, BoundingBox &box2)
{
    double cg1[2], cg2[2];
    box1.center(cg1);
    box2.center(cg2);
    double dist = sqrt(sqr(cg1[0] - cg2[0]) + sqr(cg1[1] - cg2[1]));
    return dist;
}

double BoundingBox::seperation(BoundingBox& box1)
{
    double midpoint1[2], midpoint2[2];
    midpoint1[0] = _box[2];
    midpoint1[1] = _box[1] + width()/2;
    midpoint2[0] = box1._box[0];
    midpoint2[1] = box1._box[1] + box1.width()/2;
    return sqrt(sqr(midpoint1[0] - midpoint2[0]) + sqr(midpoint1[1] - midpoint2[1]));
}

double BBox::seperation(BoundingBox& box1, BoundingBox &box2)

```

```

{
    double midpoint1[2], midpoint2[2];
    midpoint1[0] = box1._box[2];
    midpoint1[1] = box1._box[1] + box1.width()/2;
    midpoint2[0] = box2._box[0];
    midpoint2[1] = box2._box[1] + box2.width()/2;
    return sqrt(sqr(midpoint1[0] - midpoint2[0]) + sqr(midpoint1[1] - midpoint2[1]));
}

```

```

double BBox::overlap(BoundingBox &box1, BoundingBox &box2)
{
    //first check if the boxes overlap in x-direction
    double xoverlap = 0, yoverlap = 0;
    double lt, rt, tp, bt;
    if((box1._box[0] <= box2._box[0]) && (box1._box[2] >= box2._box[0]))
    {
        lt = box2._box[0];
        rt = min(box1._box[2], box2._box[2]);
    }
    else
        if((box2._box[0] <= box1._box[0]) && (box2._box[2] >= box1._box[0]))
        {
            lt = box1._box[0];
            rt = min(box1._box[2], box2._box[2]);
        }
        xoverlap = rt - lt;
#ifdef BOX
    cout << "Left : " << lt << " Right : " << rt << endl;
    cout << "Xoverlap : " << xoverlap << endl;
#endif
    //check for overlap in y-direction
    if((box1._box[1] <= box2._box[1]) && (box1._box[3] >= box2._box[1]))
    {
        bt = box2._box[1];
        tp = min(box1._box[3], box2._box[3]);
    }
    else
        if((box2._box[1] <= box1._box[1]) && (box2._box[3] >= box1._box[1]))
        {
            bt = box1._box[1];
            tp = min(box1._box[3], box2._box[3]);
        }
        yoverlap = tp - bt;
#ifdef BOX
    cout << "Top : " << tp << " Bottom : " << bt << endl;

```

```
        cout << "Yoverlap : " << yoverlap << endl;
    #endif
    return (xoverlap * yoverlap)/(box1.length()*box1.width());
}
```

```

// Camera.h: interface for the Camera class.
//
////////////////////////////////////////////////////////////////

#if
!defined(AFX_CAMERA_H__FBA75CC5_31B2_11D3_9198_0040053461F8__INCLU
DED_)
#define
AFX_CAMERA_H__FBA75CC5_31B2_11D3_9198_0040053461F8__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class Camera
{
public:
    virtual ~Camera();

    /*
    float getImageHeight() { return _ImageHeight; }
    float getPixelToCmRation() { return _PixelToCmRatio; }
    float getCameraDistance() { return _CameraDistance; }
    float getCameraFocalLength() { return _CameraFocalLength; }
    */
    void getSceneCoords(long imCoord[2], float sceneCoord[2]);
    long getReverseCoord(float z);
    static Camera& getInstance();

private:
    static Camera* _instance;
    const int _ImageHeight, _ImageWidth; //Size of frame in pixels;
    const double _Resolution; // Conversion factor for pixels to centimeters
    const int _Distance; // Distance of camera from lane 4
    const double _FocalLength; // Camera focal length (cm).
    const double _TiltAngle; // Angle of camera to the horizontal
(degrees)
    const double _PanAngle; //Rotation around Y-axis of camera
    const double _Height; // Height of camera above ground (cm);
    const double PI;

    Camera();
};

```

```
#endif //  
!defined(AFX_CAMERA_H__FBA75CC5_31B2_11D3_9198_0040053461F8__INCLU  
DED_)
```

```

// Vector2d.h: interface for the Vector2d class.
//
////////////////////////////////////

#ifndef(AFX_VECTOR2D_H__E78C93AA_55F9_11D3_916C_0040053BC61A__IN
CLUDED_)
#define
AFX_VECTOR2D_H__E78C93AA_55F9_11D3_916C_0040053BC61A__INCLUDED
-

#ifdef _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <math.h>

class Vector2d
{
public:
    Vector2d();
    Vector2d(float x, float y) { _x = x; _y = y; }
    Vector2d(float vec[2]) { _x = vec[0]; _y = vec[1]; }
    Vector2d(const Vector2d &vec) { _x = vec.getX(); _y = vec.getY(); }
    virtual ~Vector2d();
    double getX() const { return _x; };
    double getY() const { return _y; };
    void setX(double x) { _x = x; }
    void setY(double y) { _y = y; }

    double length() const { return sqrt(dotProduct(*this, *this)); }
    double angle(Vector2d &vec) const
    {
        // angle between two vectors (in radians)
        double theta = dotProduct(vec, *this)/(length()*vec.length());
        return acos(theta);
    }

    Vector2d& operator=(const Vector2d& vec)
    {
        _x = vec._x;
        _y = vec._y;
        return *this;
    }
}

```

```

void operator+=(const Vector2d &vec)
{
    _x += vec.getX();
    _y += vec.getY();
}

void operator-=(const Vector2d &vec) {
    _x -= vec.getX();
    _y -= vec.getY();
}

void operator*=(float scale)
{
    _x *= scale; _y *= scale;
}

void operator/=(float div)
{
    _x /= div; _y /= div;
}

bool operator==(const Vector2d &vec)
{
    return(_x == vec.getX() && _y == vec.getY());
}

bool operator!=(const Vector2d &vec)
{
    return(!(*this == vec));
}

private:
    double _x, _y;

    friend double dotProduct(const Vector2d &vec1, const Vector2d &vec2);
    friend double crossProduct(const Vector2d &vec1, const Vector2d &vec2);
    // vec1 X vec2

};

inline double dotProduct(const Vector2d &vec1, const Vector2d &vec2)
{
    return vec1._x * vec2._x + vec1._y * vec2._y;
}

```

```
}

inline double crossProduct(const Vector2d &vec1, const Vector2d &vec2) // vec1 X vec2
{
    return vec1._x * vec2._y - vec1._y * vec2._x;
}

Vector2d normalize(const Vector2d& vec) ;
Vector2d operator+(const Vector2d& vec1, const Vector2d& vec2);
Vector2d operator-(const Vector2d& vec1, const Vector2d& vec2);
Vector2d operator/(const Vector2d& vec1, const float div);
Vector2d operator*(const Vector2d& vec1, const float mul);

#endif //
!defined(AFX_VECTOR2D_H__E78C93AA_55F9_11D3_916C_0040053BC61A__IN
CLUDED_)
```



```

// Vehicle.h: interface for the Vehicle class.
//
////////////////////////////////////

#if
!defined(AFX_VEHICLE_H__FBA75CC1_31B2_11D3_9198_0040053461F8__INCLU
DED_)
#define
AFX_VEHICLE_H__FBA75CC1_31B2_11D3_9198_0040053461F8__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <list>
#include "blob.h"
#include "boundingBox.h"
#include "vector2d.h"
#include "blobCluster.h"
#include "Camera.h"

class VehicleMgr;

enum VehicleType { Car, Truck, Van, Pickup, SUV };

class Vehicle
{
public:
    Vehicle(Blob *blob, int name);
    Vehicle(list<Blob *> &blobs, int name);
    Vehicle(BlobCluster *_blobCluster, int name);
    virtual ~Vehicle();

    static void setManager(VehicleMgr* mgr) { _manager = mgr; }
    float getSceneLength() { return _sceneBBox.length(); }
    float getSceneWidth() { return _sceneBBox.width(); }
    float getImgLength() { return _blobCluster->getLength(); }
    float getImgWidth() { return _blobCluster->getWidth(); }
    static bool isValidVehicle(Blob *blob);
    bool canMerge(BlobCluster& bclust);

    void show();
    void update();
    void removeBlob(Blob *blob);

protected:

```

```

void _replaceBlobs(BlobCluster *blobs);
void _setType(enum VehicleType type) { _type = type; }
void _createVehicle(int name);

Vector2d _sceneVelocity;
BoundingBox _sceneBBox;
enum VehicleType _type;
int _name;
BlobCluster *_blobCluster;
static VehicleMgr* _manager;
bool _zombie;
int _age;
static float const _MinVehicleLength, _MaxVehicleLength,
                 _MinVehicleWidth, _MaxVehicleWidth,
                 _InitialVelocityX, _InitialVelocityY;

static VisionProcessor* _visProc;

friend class VehicleMgr;

};

namespace Veh {
    BoundingBox calcSceneCoords(BoundingBox &box);
};

#endif //
!defined(AFX_VEHICLE_H__FBA75CC1_31B2_11D3_9198_0040053461F8__INCLU
DED_)

```

```

// VehicleClassifier.h: interface for the VehicleClassifier class.
//
/////////////////////////////////////////////////////////////////

#if
!defined(AFX_VEHICLECLASSIFIER_H__6E63EEC2_45DE_11D3_9159_004005346
1F8__INCLUDED_)
#define
AFX_VEHICLECLASSIFIER_H__6E63EEC2_45DE_11D3_9159_0040053461F8__IN
CLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "vehicle.h"

class VehicleClassifier
{
public:
    VehicleClassifier();
    virtual ~VehicleClassifier();

    enum VehicleType classify(Vehicle &veh);
    const int _CarWidth;

};

#endif //
!defined(AFX_VEHICLECLASSIFIER_H__6E63EEC2_45DE_11D3_9159_004005346
1F8__INCLUDED_)

```

```

// VehicleMgr.h: interface for the VehicleMgr class.
//
////////////////////////////////////

#if
!defined(AFX_VEHICLEMGR_H__6E63EEC1_45DE_11D3_9159_0040053461F8__I
NCLUDED_)
#define
AFX_VEHICLEMGR_H__6E63EEC1_45DE_11D3_9159_0040053461F8__INCLUDE
D_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "blob.h"
#include "Vehicle.h"
#include "VehicleClassifier.h"
#include <list>

class VehicleMgr
{
public:
    VehicleMgr();
    virtual ~VehicleMgr();

    void setClusterer(BlobClusterer *clusterer) { _clusterer = clusterer; }
    void createVehicle(BlobCluster const *blobCluster); // should be a list of
blobs
    void showVehicles();
    void update();
    void findMissingBlobs(Vehicle *veh);
    bool canCreateVehicle(Blob* blob);

private:
    Vehicle* _isNewVehicle(Vehicle *vh) ;
    bool _isUniqueVehicle(Vehicle *vh);
    void _createUniqueVehicle(Blob *blob);
    list<Vehicle*> _vehicles;
    list<Vehicle*> _incompleteClusters; // blobClusters from vehicles that have
requested blobs
    VehicleClassifier _vehicleClassifier;
    BlobClusterer *_clusterer;
    static int _count;
};

```

```
#endif //  
!defined(AFX_VEHICLEMGR_H__6E63EEC1_45DE_11D3_9159_0040053461F8__I  
NCLUDED_)
```

```

// VisionProcessor.h: interface for the VisionProcessor class.
//
////////////////////////////////////

#if
!defined(AFX_VISIONPROCESSOR_H__CA4842B1_42CF_11D3_919A_0040053461
F8__INCLUDED_)
#define
AFX_VISIONPROCESSOR_H__CA4842B1_42CF_11D3_919A_0040053461F8__INC
LUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <vector>
#include <list>

#include "imapi.h"
#include "blobData.h"

class VisionProcessor
{
#define MAX_BLOBS 20

public:
    static VisionProcessor* getInstance();
    virtual ~VisionProcessor();

    int grabFrame();
    void initGrab();

    void segmentImage(int frame_no);
    void calculateBlobs();
    std::list<blobData*>* getBlobs();
    int getBlobCount() { return _count; }

    void showBlob(const blobData &bdata, int name);
    void showVehicle(long *bBox, enum VehicleType type, int name);

    void startClock() { _start = imSysClock(_thread, 0); }
    double stopClock() { return imSysClock(_thread, _start); }

    int getNumFrames() { return _frames; }

```

```

private:
    const int _NumBuffers;          // Number of buffers in buffer pool

    VisionProcessor();
//    int _next(int i) { return (i++)%_NumBuffers; }
//    int _prev(int i) { return (i+_NumBuffers-1)%_NumBuffers; }

    int _next(int i) { return (i+1)%3; }
    int _prev(int i) { return (i+2)%3; }

    static VisionProcessor* _instance;
    long _dispBuff[2], _resBuff, _blobBuff;    // Display buffers
    long _grabPBuff[3]; // Buffer pool
    long _grabOSB[3]; // OSB for the grab buffers
    long _copyOSB;
    long _edgePBuff, _procPBuff, _imPBuff[2], _ePBuff; // processing buffers
    long _idPBBuff, _edgePBBuff[2], _procPBBuff; // processing binary
buffers
    long _digControl; // Control buffer for camera
    long _coeffBuff; // Control buffer for Warp function
    long _count; // Number of blobs
    long _thread; // For now just one thread, should support multiple threads
    long _camera; // handle to the camera
    long _device; // handle to the Genesis device
    long _initFeature, _feature;
    long _result;

    long _morphControl;
    long _graControl;
    long _textControl;
    long _procBuff, _idBBuff, _blobBBuff; // Processing buffers, the B are
binary buffers

    //arrays for getting results of blob calculation
    long _minX[MAX_BLOBS], _minY[MAX_BLOBS], _maxX[MAX_BLOBS],
_maxY[MAX_BLOBS];
    long _area[MAX_BLOBS];
    long _cgX[MAX_BLOBS], _cgY[MAX_BLOBS];
    long _label[MAX_BLOBS];

    double _start; // Start of clock
//Constants for the size filter
    const int _InitMinLength, _InitMaxLength,
        _BlobMinLength, _BlobMaxLength;

    int _sizeX, _sizeY;

```

```
int _frames, _grabIndex, _procIndex;
// Subtraction is done these many frames apart
int _frameDiff;

// Difference in seconds between the two frames that are differenced
float _timeDiff;

//flag to indicate if results have been transferred to host (lazy transfer)
bool _resultTx;
};

#endif //
!defined(AFX_VISIONPROCESSOR_H__CA4842B1_42CF_11D3_919A_0040053461
F8__INCLUDED_)
```



```

// BlobCluster.cpp: implementation of the BlobCluster class.
//
////////////////////////////////////////////////////////////////

#include "BlobCluster.h"
#include "BlobManager.h"
#include "Vehicle.h"
#include <math.h>
#include <stdexcept>
#include <iostream>

using std::list;
using std::cout;
using std::ostream;
using std::endl;

const float BlobCluster::_MaxClusterLength = 500; // Should be same as
Vehicle::MaxLength
const float BlobCluster::_MaxClusterWidth = 200;
const float BlobCluster::_VelocityUpdateFactor = 0.5;
const float BlobCluster::_MaxBlobClusterDist = 50;
BlobManager& BlobCluster::_blobMgr = BlobManager::getInstance();

////////////////////////////////////////////////////////////////
// Construction/Destruction
////////////////////////////////////////////////////////////////

BlobCluster::BlobCluster() : _box(0,0,0,0)
{
}

BlobCluster::BlobCluster(Blob *blob) : _box(blob->getBoundingBox())
{
    _blobs.push_back(blob);
    _length = _box.length();
    _width = _box.width();
    _imgVelocity = blob->getVelocity();
    _replacing = false;
}

BlobCluster::~BlobCluster()
{
    _blobMgr.removeBlobs(_blobs);
    _blobs.erase(_blobs.begin(), _blobs.end());
}

```

```

}

void BlobCluster::replaceBlobs(BlobCluster* blobs)
{
    //dirty little hack
    _replacing = true;
    // painfully inefficient, but it's just for now (yeah, right!!)
    for(std::list<Blob*>::iterator iter = _blobs.begin(); iter != _blobs.end(); iter++)
    {
#ifdef BLOBCLUSTER
        cout << "Removing blob " << (*iter)->getName() << endl;
#endif
        _blobMgr.removeBlob(*iter);
    }
    _blobs.erase(_blobs.begin(), _blobs.end());
    _blobs.insert(_blobs.begin(), blobs->_blobs.begin(), blobs->_blobs.end());
    updateDimensions();
    _replacing = false;
}

void BlobCluster::assignVehicle(Vehicle *veh)
{
    for(std::list<Blob*>::iterator iter = _blobs.begin(); iter != _blobs.end(); iter++)
    {
        (*iter)->setVehicle(veh);
    }
}

void BlobCluster::updateDimensions()
{
    Blob *blob;
    std::list<Blob*>::const_iterator citor = _blobs.begin();
    blob = *citor;
    if(!_blobs.size())
    {
//        _box.setCoordinates(0,0,0,0);
        return;
    }

    Vector2d vel(blob->getVelocity());
    _box = (*citor)->getBoundingBox();
    for(; citor != _blobs.end(); citor++)
    {
        blob = *citor;
        _box += blob->getBoundingBox();
    }
}

```

```

        vel += blob->getVelocity();
    }
    // Velocity of BlobCluster is average velocity of it's component Blobs
    vel = vel/_blobs.size();
    _imgVelocity = vel*_VelocityUpdateFactor + _imgVelocity*(1 -
_VelocityUpdateFactor);
}

void BlobCluster::removeBlob(Blob *blob)
{
    if(!_replacing)
        return;
    for(std::list<Blob*>::iterator iter = _blobs.begin(); iter != _blobs.end(); iter++)
    {
        if(*iter == blob)
        {
            _blobs.erase(iter);
            break;
        }
    }
    updateDimensions();
}

bool BlobCluster::_merge(Cluster &cluster)
{
    if(!_canBeMerged(cluster))
        return false;

    Vehicle *veh = (*_blobs.begin())->getVehicle();
    try {
        BlobCluster &bClust= dynamic_cast<BlobCluster&>(cluster);
#ifdef BLOBCLUSTER
        cout << "BlobCluster(" << *this << ") merged with (" << bClust <<
        ")n";
#endif
        bClust.assignVehicle(veh);
        _blobs.insert(_blobs.end(), bClust._blobs.begin(), bClust._blobs.end());
        _box += bClust._box;
        _imgVelocity = (_imgVelocity + bClust._imgVelocity)/2; // This is wrong
    }
    catch(std::bad_cast &b) {
        cout << "Object not of type BlobCluster&n";
        exit(-1);
    }
    return true;
}

```

```

}

double BlobCluster::_similarity(Cluster &cluster) {
    try {
        BlobCluster &bClust = dynamic_cast<BlobCluster&>(cluster);
        double dist = _box.distance(bClust.getBoundingBox());
        return 1/dist;
    }
    catch(std::bad_cast &b) {
        cout << "Object not of type BlobCluster&\n";
        exit(-1);
    }
}

double BlobCluster::_distance(Cluster &cluster) {
    try {
        BlobCluster &bClust = dynamic_cast<BlobCluster&>(cluster);
        return _box.distance(bClust.getBoundingBox());
    }
    catch(std::bad_cast &b) {
        cout << "Object not of type BlobCluster&\n";
        exit(-1);
    }
}

bool BlobCluster::_canBeMerged(Cluster &cluster) {

    try {
        BlobCluster &bClust = dynamic_cast<BlobCluster&>(cluster);

        //two BlobClusters can be merged if i) the differences in their velocities
        // is not too great, ii) The combined length of the two, does not exceed the
        // maximum vehicle length and width.
        BoundingBox bbox = bClust.getBoundingBox();
        float dist = _box.distance(bbox) - (bbox.width() + _box.width())/2;
        if(dist > _MaxBlobClusterDist)
            return false;
        // double speedDiff = fabs(_imgVelocity.length() -
        bClust._imgVelocity.length());
        // double theta = _imgVelocity.angle(bClust._imgVelocity);
        // if(speedDiff >= 0.5*_imgVelocity.length() || (theta >= 0.5))
        // return false;
        bool res = (*_blobs.begin()->getVehicle()->canMerge(bClust);
        if(res) {
#ifdef BLOBCLUSTER

```

```

        std::cout << "Combined length : " << length << " : " << width <<
endl;
#endif
        return true;
    }
    return false;
}
catch(std::bad_cast b) {
    cout << "Object not of type BlobCluster&\n";
    exit(-1);
}
}

ostream& operator <<(ostream &ostr, BlobCluster& cluster)
{
    for(std::list<Blob*>::const_iterator iter = cluster._blobs.begin(); iter !=
cluster._blobs.end(); iter++)
    {
        ostr << (*iter)->getName() << " ";
    }
    return ostr;
}

ostream& operator <<(ostream &ostr, Cluster& cluster)
{
    try {
        BlobCluster &bClust = dynamic_cast<BlobCluster&>(cluster);
        return ostr<< bClust;
    }
    catch(std::bad_cast b) {
        cout << "Object not of type BlobCluster&\n";
        exit(-1);
    }
}
}

```

```

// BlobClusterer.cpp: implementation of the BlobClusterer class.
//
////////////////////////////////////

#include "BlobClusterer.h"
#include "BoundingBox.h"
#include <iostream>
#include <conio.h>

using std::cout;
using std::endl;

////////////////////////////////////
// Construction/Destruction
////////////////////////////////////

BlobClusterer::BlobClusterer(VehicleMgr &vMgr) : _vehicleMgr(vMgr)
{
}

BlobClusterer::~BlobClusterer()
{
}

void BlobClusterer::isolateOrphans(list <Blob*>& blobs)
{
    // Just treat each blob as one vehicle
    Blob *blob;

    _orphanBlobs.erase(_orphanBlobs.begin(), _orphanBlobs.end());
    for(std::list<Blob*>::const_iterator itor = blobs.begin(); itor != blobs.end();
itor++)
    {
        blob = *itor;
        //isVehicle creates a new vehicle if isVehicle is true
        if(!(blob->_vehicle || _vehicleMgr.canCreateVehicle(blob)))
            _orphanBlobs.push_back((Cluster*) new BlobCluster(blob));

        // Since Clusterer expects a list<Cluster*>, therefore typecast
        //BlobCluster* to Cluster*, we retrieve the type using RTTI
        // in BlobCluster
    }
}

```

```

void BlobClusterer::clusterOrphans()
{
    list <BlobCluster*> cluster;

    //    _clusterer.hierarchicalCluster(_orphanBlobs);
}

void BlobClusterer::findMatchingBlobs(BlobCluster& bCluster) {
    _clusterer.expandCluster(bCluster, _orphanBlobs);
}

void BlobClusterer::findMatchingBlobs(list<BlobCluster*>& bCluster) {
    list<Cluster*> clusters;
    for(std::list<BlobCluster*>::iterator iter = bCluster.begin(); iter != bCluster.end();
iter++)
        clusters.push_back((Cluster*)*iter);

    _clusterer.expandClusters(clusters, _orphanBlobs);
}

```

```

// Clusterer.cpp: implementation of the Clusterer class.
//
////////////////////////////////////////////////////////////////

#include "Clusterer.h"
#include <vector>
#include <iostream>

using namespace std;

template<class T> void showList(std::list<T> &li);

////////////////////////////////////////////////////////////////
// Construction/Destruction
////////////////////////////////////////////////////////////////

Clusterer::Clusterer()
{

}

Clusterer::~Clusterer()
{

}

bool Clusterer::expandCluster(Cluster &cluster, list<Cluster*> &clusters)
{
    return expandCluster(cluster, clusters, clusters.begin());
}

bool Clusterer::expandCluster(Cluster &cluster, list<Cluster*> &clusters,
                               std::list<Cluster*>::iterator start)
{
    float similarity, maxSimilarity = 0;
    std::list<Cluster*>::iterator mostSimilarCluster = clusters.end();

    for(std::list<Cluster*>::iterator iter = start; iter != clusters.end(); iter++)
    {
        similarity = cluster._similarity(**iter);
        if(similarity > maxSimilarity && cluster._canBeMerged(**iter))
        {
            maxSimilarity = similarity;
            mostSimilarCluster = iter;
        }
    }
}

```



```

    }
    if(mostSimilarCluster != clusters.end())
    {
        cluster._merge(**mostSimilarCluster);
        delete *mostSimilarCluster;
        clusters.erase(mostSimilarCluster);
        return true;
    }
    return false;
}

void Clusterer::expandClusters(list<Cluster*>& clusters, list<Cluster*>&clusterees)
{
    float sim, max_sim = 0;
    int i = 0, j = 0;

    // sim_vector is an array of lists. There is one entry in the array for each cluster
    // This entry is a list of clusterees that could potentially be clustered with this
cluster
    vector<list<Cl> > sim_vector(clusters.size());

    for(std::list<Cluster*>::iterator clusteree_iter = clusterees.begin();
        clusteree_iter != clusterees.end(); clusteree_iter++)
    {
        max_sim = 0;
        i = 0;
        for(std::list<Cluster*>::iterator cliter = clusters.begin();
            cliter != clusters.end(); cliter++, i++)
        {
            sim = (*cliter)->_similarity(**clusteree_iter);
            if(sim > max_sim)
            {
                max_sim = sim;
                j = i;
            }
        }
        sim_vector[j].push_back(Cl(max_sim, cliter, clusteree_iter));
    }

    int size = clusters.size();
    i = 0;
    std::greater<Cl> c;

    for(std::list<Cluster*>::iterator iter = clusters.begin();

```

```

        i < size; i++, iter++)
    {
#ifdef CLUSTER
        std::cout << "Trying to merge cluster" << **iter << endl;
#endif

        std::list<Cl>::const_iterator citer = sim_vector[i].begin();
        sim_vector[i].sort(c);

        int sz = (sim_vector[i]).size();
        for(citer = sim_vector[i].begin();
            citer != (sim_vector[i]).end(); citer++) {
            // Cluster.merge() returns false if it cannot be merged with the
given cluster
#ifdef CLUSTER
            std::list<Cluster*>::iterator it = (*citer).clusteree;
            std::cout << "\t with " << **it ;
#endif

            if((*iter)->_merge(**it))
            {
                //delete *citer;
                clusterees.erase(it);
            }
#ifdef CLUSTER
            else
            {
                std::cout << "\tCould not merge clusters\n";
            }
#endif
        }
    }
}

template<class T> void showList(list<T> &li) {
    for(list<T>::iterator iter = li.begin(); iter != li.end(); iter++)
        cout << *iter;
}

#ifdef CLUSTER
std::ostream& operator<< (ostream &ostr, Clusterer::Cl& c1) {
    std::list<Cluster*>::iterator cluster = c1.cluster;
    std::list<Cluster*>::iterator clusteree = c1.clusteree;
    ostr << "(" << (**cluster) << " ) " << "(" << (**clusteree) << " ) " <<
c1.similarity << endl;
}

```

```
        return ostr;  
    }  
#endif
```

```

// Blob.cpp: implementation of the Blob class.
//
////////////////////////////////////

#include "Blob.h"
#include "vehicle.h"
#include "VisionProcessor.h"
#include <iostream>
#include <math.h>
#include "imapi.h"

VisionProcessor* Blob::_visProc = 0;
int Blob::_count = 0;

////////////////////////////////////
// Construction/Destruction
////////////////////////////////////

Blob::~Blob()
{
#ifdef BLOBB
    std::cout << "Blob " << _name << " erased " << std::endl;
#endif //BLOB

    if(_vehicle)
        _vehicle->removeBlob(this);
}

Blob::Blob(blobData &bdata) : _blobData(bdata), _boundingBox(bdata.boundingBox),
    _velocity(0,0) {
    if(!_visProc)
        _visProc = VisionProcessor::getInstance();
    _name = _count++;
    _vehicle = 0;
#ifdef BLOBB
    std::cout << "Blob " << _name << " created \n";
#endif
}

int Blob::distance(long cg[2]) {
    int distX = _blobData.centerGravity[0] - cg[0];
    int distY = _blobData.centerGravity[1] - cg[1];
    return (sqrt(distX*distX + distY*distY));
}

```

```
void Blob::update(blobData &bd) {
    //velocity in pixels/frame
    //    _velocity.setX((bd.centerGravity[0] - _blobData.centerGravity[0])/_visProc-
    >frameDiff());
    //    _velocity.setY((bd.centerGravity[1] - _blobData.centerGravity[1])/_visProc-
    >frameDiff());
    _blobData = bd;
    _boundingBox.setCoordinates(bd.boundingBox);
}

void Blob::show() {
    _visProc->showBlob(_blobData, _name);
}
```

```

// BlobManager.cpp: implementation of the BlobManager class.
//
////////////////////////////////////

#include "BlobManager.h"
#include "visionProcessor.h"
#include <iostream>

using std::cout;
using std::endl;
using std::list;

const unsigned int BlobManager::_MinBlobDistance = 50;
const unsigned int BlobManager::_MinBlobDisplacement = 2;
const unsigned int BlobManager::_MaxStaticCount = 2;
const unsigned int BlobManager::_MaxAge = 2;
const float BlobManager::_OverlapThreshold = 0.5;
BlobManager* BlobManager::_instance;

////////////////////////////////////
// Construction/Destruction
////////////////////////////////////

BlobManager::BlobManager()
{
}

BlobManager::~BlobManager()
{
}

BlobManager& BlobManager::getInstance()
{
    if(!_instance)
        _instance = new BlobManager();
    return *_instance;
}

#if 0
void BlobManager::addBlobs(list<blobData*> &blobList) {
    int dist,minDist;
    float sizeDiff;
    std::list<blobData*>::iterator match;
}
#endif

```

```

//Simplistic matching of blobs, just using a list O(sqr(n))

for(std::list<Blob*>::iterator iter = _blobs.begin(); iter != _blobs.end(); iter++) {
    minDist = _MinBlobDistance;
    match = blobList.end();
#ifdef BLOB_MGR
    cout << "Matching blob " << (*iter)->_name << endl;
#endif

    //remove old blobs
    if((*iter)->_age > _MaxAge) {
#ifdef BLOB_MGR
        cout << (*iter)->_name << " too old\n";
#endif

        delete *iter;
        iter = _blobs.erase(iter);
    }
    else {
        //remove blobs that haven't moved for a long time;
        if((*iter)->_staticCount > _MaxStaticCount) {

#ifdef BLOB_MGR
            cout << (*iter)->_name << " immobile\n";
#endif

            delete *iter;
            iter = _blobs.erase(iter);
        }
    }
    if(iter == _blobs.end())
        break;

    //Compare this blob with each blob from the new list of blobData
    for(std::list<blobData*>::iterator bdItor = blobList.begin(); bdItor !=
blobList.end(); bdItor++) {
        //instead of considering distance, to search for a matching blob,
        //consider overlap

        /* if there is significant overlap (say 90%), check for 3 cases
        1. The size of the blob(t-1) (blob in previous frame) and
blob(t)
            (blob in current frame) is approximately equal
            Match, no splitting or merging has occurred
        2. blob(t-1) smaller than blob(t)
            blob(t-1) merged with other blobs to form
blob(t)
        3. blob(t-1) larger than blob(t)

```

```

blob
                                blob(t-1) split into blob(t) and some other
                                */
                                dist = (*iter)->distance((*bdItor)->centerGravity);
//distance to this blob
                                if(dist <= minDist) {
                                    minDist = dist;
                                    match = bdItor;
                                }
                                }
                                //to be considered a match, the size of the new blob should not differ
                                //by more than 50% of previous blob
                                if(match != blobList.end()) {
                                    sizeDiff = abs((*iter)->getArea() - (*match)->area); // Difference
in size
                                    if(sizeDiff <= 0.5 * (*iter)->getArea()) {
                                        // Yes, finally we have a match
#ifdef BLOB_MGR
                                        cout << (*iter)->_name << " matched\n";
#endif
                                        (*iter)->update(**match);
                                        delete *match;
                                        blobList.erase(match);

                                        //blob matched, but did not show "significant" movement
                                        if(minDist < _MinBlobDisplacement)
                                            (*iter)->_staticCount++;
                                    }
                                    else {
#ifdef BLOB_MGR
                                        cout << (*iter)->_name << " not matched" << endl;
#endif
                                        (*iter)->incAge();
                                    }
                                }
                                // This blob didn't match any of the blobs in the current frame
                                else {
                                    //increment age for all the blobs that haven't been matched;
                                    (*iter)->incAge();
#ifdef BLOB_MGR
                                    cout << (*iter)->_name << " not matched" << endl;
//
                                    cout << "Best match : " << minDist << " " << sizeDiff/ << endl;
#endif
                                }
                            }
                        }

```



```

        for(std::list<blobData*>::iterator bdItor = blobList.begin(); bdItor !=
blobList.end(); bdItor++) {
            Blob* bl = new Blob(**bdItor);
            _blobs.insert(_blobs.begin(), bl);
            bl->show();
        }
    }
}

#endif

/* The BlobManager is a given a list of blobData's. Matches the blobs in the current
frame
** to those in the previous frame. In this version of addBlobs, the BlobManager simply
removes
** blobs that are not matched
*/

void BlobManager::addBlobs(list<blobData*> &blobList) {
    float overlap, max_overlap;
    float sizeDiff;
    std::list<blobData*>::iterator match;
    std::list<Blob*>::iterator last = _blobs.end();

    //Simplistic matching of blobs, just using a list O(sqr(n))
    for(std::list<Blob*>::iterator iter = _blobs.begin(); iter != _blobs.end(); iter++) {
        max_overlap = 0;
        match = blobList.end();

        //Compare this blob with each blob from the list of new BlobData

        for(std::list<blobData*>::iterator bdItor = blobList.begin(); bdItor !=
blobList.end(); bdItor++) {
            // symOverlap gives the maximum of the overlap of the blob with
blobData and overlap
            // of blobData with blob
            //
            // overlap = (*iter)-
            // >_boundingBox.symOverlap(BoundingBox((*bdItor)->boundingBox));
            // overlap = (*iter)->_boundingBox.overlap(BoundingBox((*bdItor)-
            // >boundingBox));
            if(overlap > max_overlap) {
                max_overlap = overlap;
                match = bdItor;
            }
        }
        if(max_overlap > _OverlapThreshold) {

```

```

        // Yes, finally we have a match
        (*iter)->update(**match);
        delete *match;
        blobList.erase(match);
#ifdef BLOB_MGR
        cout << (*iter)->_name << " matched\n";
#endif
    }
    // This blob didn't match any of the blobs in the current frame
    else {
#ifdef BLOB_MGR
        cout << (*iter)->_name << " not matched" << endl;
#endif
        delete *iter;
        iter = _blobs.erase(iter);
        iter--;
    }
}

//create new blobs for all the unmatched blobData's
for(std::list<blobData*>::iterator bdItor = blobList.begin(); bdItor !=
blobList.end(); bdItor++) {
    _blobs.push_back(new Blob(**bdItor));
}

void BlobManager::showBlobs() {
    for(std::list<Blob*>::const_iterator iter = _blobs.begin(); iter != _blobs.end();
iter++) {
        (*iter)->show();
    }
}

void BlobManager::showMatchedBlobs() {
    for(std::list<Blob*>::const_iterator iter = _blobs.begin(); iter != _blobs.end();
iter++) {
        (*iter)->show();
    }
}

void BlobManager::removeBlob(Blob* blob)
{
    for(std::list<Blob*>::iterator iter = _blobs.begin(); iter != _blobs.end(); iter++)
    {
        if(blob->_name == (*iter)->_name)

```

```
        {
            delete *iter;
            iter = _blobs.erase(iter);
            iter--;
        }
    }

void BlobManager::removeBlobs(list<Blob*>& blobs)
{
}
}
```

```

// BoundingBox.cpp: implementation of the BoundingBox class.
//
////////////////////////////////////

#include "BoundingBox.h"
#include <math.h>
#include <iostream>

using std::cout;
using std::endl;

inline double sqr(double x) { return (x) * (x); }
inline double max(double x1, double x2) { return ( (x1) > (x2) ? (x1) : (x2)); }
inline double min(double x1, double x2) { return ( (x1) < (x2) ? (x1) : (x2)); }

////////////////////////////////////
// Construction/Destruction
////////////////////////////////////

BoundingBox::BoundingBox(double left, double bottom, double right, double top)
{
    _box[0] = left; _box[1] = bottom;
    _box[2] = right; _box[3] = top;
}

BoundingBox::BoundingBox(double box[4])
{
    _box[0] = box[0]; _box[1] = box[1];
    _box[2] = box[2]; _box[3] = box[3];
}

BoundingBox::BoundingBox(long box[4])
{
    _box[0] = box[0]; _box[1] = box[1];
    _box[2] = box[2]; _box[3] = box[3];
}

BoundingBox::~BoundingBox()
{
}

void BoundingBox::operator+= (BoundingBox &bBox)
{
    double *box = bBox.coordinates();
    if(box[0] < _box[0] )

```

```

        _box[0] = box[0];
    if(box[1] < _box[1] )
        _box[1] = box[1];
    if(box[2] > _box[2] )
        _box[2] = box[2];
    if(box[3] > _box[3] )
        _box[3] = box[3];
}

void BoundingBox::center(double cg[])
{
    cg[0] = _box[0] + length()/2;
    cg[1] = _box[1] + width()/2;
}

double BoundingBox::symOverlap(BoundingBox &box1) {
    double ovr1 = overlap(box1);
    double ovr2 = box1.overlap(*this);
    return ovr1 > ovr2 ? ovr1 : ovr2;
}

double BoundingBox::overlap(BoundingBox &box1) {
    //first check if the boxes overlap in x-direction
    double xoverlap = 0, yoverlap = 0;
    double lt, rt, tp, bt;
    if((_box[0] <= box1._box[0]) && (_box[2] >= box1._box[0]))
    {
        lt = box1._box[0];
        rt = min(_box[2], box1._box[2]);
    }
    else
        if((box1._box[0] <= _box[0]) && (box1._box[2] >= _box[0]))
        {
            lt = _box[0];
            rt = min(_box[2], box1._box[2]);
        }
        xoverlap = rt - lt;
#ifdef BOX
    cout << "Left : " << lt << " Right : " << rt << endl;
    cout << "Xoverlap : " << xoverlap << endl;
#endif
}

//check for overlap in y-direction
if((_box[1] <= box1._box[1]) && (_box[3] >= box1._box[1]))
{
    bt = box1._box[1];
    tp = min(_box[3], box1._box[3]);
}

```

```

    }
    else
        if((box1._box[1] <= _box[1]) && (box1._box[3] >= _box[1]))
        {
            bt = _box[1];
            tp = min(_box[3], box1._box[3]);
        }
        yoverlap = tp - bt;

#ifdef BOX
    cout << "Top : " << tp << " Bottom : " << bt << endl;
    cout << "Yoverlap : " << yoverlap << endl;
#endif
    return (xoverlap * yoverlap)/(length()*width());
}

double BoundingBox::distance(BoundingBox &box1)
{
    double cg1[2], cg2[2];
    center(cg1);
    box1.center(cg2);
    double dist = sqrt(sqr(cg1[0] - cg2[0]) + sqr(cg1[1] - cg2[1]));
    return dist;
}

double BBox::distance(BoundingBox &box1, BoundingBox &box2)
{
    double cg1[2], cg2[2];
    box1.center(cg1);
    box2.center(cg2);
    double dist = sqrt(sqr(cg1[0] - cg2[0]) + sqr(cg1[1] - cg2[1]));
    return dist;
}

double BoundingBox::seperation(BoundingBox& box1)
{
    double midpoint1[2], midpoint2[2];
    midpoint1[0] = _box[2];
    midpoint1[1] = _box[1] + width()/2;
    midpoint2[0] = box1._box[0];
    midpoint2[1] = box1._box[1] + box1.width()/2;
    return sqrt(sqr(midpoint1[0] - midpoint2[0]) + sqr(midpoint1[1] - midpoint2[1]));
}

double BBox::seperation(BoundingBox& box1, BoundingBox &box2)

```

```

{
    double midpoint1[2], midpoint2[2];
    midpoint1[0] = box1._box[2];
    midpoint1[1] = box1._box[1] + box1.width()/2;
    midpoint2[0] = box2._box[0];
    midpoint2[1] = box2._box[1] + box2.width()/2;
    return sqrt(sqr(midpoint1[0] - midpoint2[0]) + sqr(midpoint1[1] - midpoint2[1]));
}

```

```

double BBox::overlap(BoundingBox &box1, BoundingBox &box2)
{
    //first check if the boxes overlap in x-direction
    double xoverlap = 0, yoverlap = 0;
    double lt, rt, tp, bt;
    if((box1._box[0] <= box2._box[0]) && (box1._box[2] >= box2._box[0]))
    {
        lt = box2._box[0];
        rt = min(box1._box[2], box2._box[2]);
    }
    else
        if((box2._box[0] <= box1._box[0]) && (box2._box[2] >= box1._box[0]))
        {
            lt = box1._box[0];
            rt = min(box1._box[2], box2._box[2]);
        }
        xoverlap = rt - lt;
#ifdef BOX
    cout << "Left : " << lt << " Right : " << rt << endl;
    cout << "Xoverlap : " << xoverlap << endl;
#endif
    //check for overlap in y-direction
    if((box1._box[1] <= box2._box[1]) && (box1._box[3] >= box2._box[1]))
    {
        bt = box2._box[1];
        tp = min(box1._box[3], box2._box[3]);
    }
    else
        if((box2._box[1] <= box1._box[1]) && (box2._box[3] >= box1._box[1]))
        {
            bt = box1._box[1];
            tp = min(box1._box[3], box2._box[3]);
        }
        yoverlap = tp - bt;
#ifdef BOX
    cout << "Top : " << tp << " Bottom : " << bt << endl;

```

```
        cout << "Yoverlap : " << yoverlap << endl;
    #endif
    return (xoverlap * yoverlap)/(box1.length()*box1.width());
}
```



```

// Camera.cpp: implementation of the Camera class.
//
////////////////////////////////////

#include "Camera.h"
#include <iostream>
#include <math.h>

using std::cout;
using std::endl;

Camera* Camera::_instance;

////////////////////////////////////
// Construction/Destruction
////////////////////////////////////

Camera::Camera() : _ImageHeight(240), _ImageWidth(320), _Resolution(1000),
    _Distance(686),
    _FocalLength(0.689), _TiltAngle(-39.54), _PanAngle(-15.0),
    _Height(977.35), PI(3.1415926535897932384626433832795)
{
}

Camera::~~Camera()
{
}

Camera& Camera::getInstance() {
    if(!_instance)
        _instance = new Camera;
    return *_instance;
}

void Camera::getSceneCoords(long imCoord[2], float sceneCoord[2]) {
    //convert 3rd quadrant coords to first quadrant coords
    float z;
    float y = (_ImageHeight/2 - imCoord[1])/_Resolution;    // everything has to be
in cm
    float x = (imCoord[0] - _ImageWidth/2)/_Resolution;
    float alpha = PI*_TiltAngle/180;    //radians
    float beta = PI*_PanAngle/180;

//    cout << "Y"

```

```

    z = _Height*(_FocalLength - y*tan(alpha))/(y + _FocalLength*tan(alpha));

//    cout << "\tDistance : " << z << endl;
//    cout << "\t" << sceneCoord[0] << "\t" << sceneCoord[1] << endl;

//Perspective projection equation
sceneCoord[0] = x*(-z)/(_FocalLength*cos(beta));
// sceneCoord[1] = y*(-z)/_FocalLength;
sceneCoord[1] = fabs(z);
}

long Camera::getReverseCoord(float z) {
    float gamma = atan(_Height/z);
    float delta = gamma - _TiltAngle*PI/180;
    float zc = z*cos(delta)/cos(gamma);
    float yc = z*sin(delta)/cos(gamma);
    float y = _FocalLength*yc/zc;           // Perspective equation;
    y = y*_Resolution;                   // Convert to pixels
    return (long)y;
}

```

```

// Vector2d.cpp: implementation of the Vector2d class.
//
/////////////////////////////////////////////////////////////////

#include "Vector2d.h"

/////////////////////////////////////////////////////////////////
// Construction/Destruction
/////////////////////////////////////////////////////////////////

Vector2d::Vector2d()
{
}

Vector2d::~Vector2d()
{
}

Vector2d normalize(const Vector2d& vec)
{
    double len = vec.length();
    Vector2d vect(vec.getX()/len, vec.getY()/len);
    return vect;
}

Vector2d operator+(const Vector2d& vec1, const Vector2d& vec2)
{
    Vector2d vect(vec1.getX(), vec1.getY());
    vect += vec2;
    return vect;
}

Vector2d operator-(const Vector2d& vec1, const Vector2d& vec2)
{
    Vector2d vect(vec1.getX(), vec1.getY());
    vect -= vec2;
    return vect;
}

Vector2d operator/(const Vector2d& vec1, const float div)
{
    Vector2d vect(vec1.getX(), vec1.getY());
    vect /= div;
    return vect;
}

```

```
}  
  
Vector2d operator*(const Vector2d& vec1, const float mul)  
{  
    Vector2d vect(vec1.getX(), vec1.getY());  
    vect *= mul;  
    return vect;  
}
```

```

// Vehicle.cpp: implementation of the Vehicle class.
//
////////////////////////////////////////////////////////////////

#include "Vehicle.h"
#include "visionProcessor.h"
#include "vehicleMgr.h"
#include <iostream>

////////////////////////////////////////////////////////////////
// Construction/Destruction
////////////////////////////////////////////////////////////////
const float Vehicle::_InitialVelocityX = 20;
const float Vehicle::_InitialVelocityY = 20;
VisionProcessor* Vehicle::_visProc = 0;

using std::cout;
using std::endl;
using Veh::calcSceneCoords;

Vehicle::Vehicle(BlobCluster *blobCluster, int name) : _zombie(false),
    _sceneVelocity(_InitialVelocityX, _InitialVelocityY)
{
    _blobCluster = blobCluster;
    _createVehicle(name);
}

Vehicle::Vehicle(Blob *blob, int name) : _zombie(false),
    _sceneVelocity(_InitialVelocityX, _InitialVelocityY)
{
    _blobCluster = new BlobCluster(blob);
    _createVehicle(name);
}

Vehicle::~Vehicle()
{
    //This will have to be taken care of in ~BlobCluster

#ifdef VEHICLE
    std::cout << "Vehicle " << _name << " removed" << std::endl;
#endif
}

void Vehicle::_createVehicle(int name) {

```

```

    if(!_visProc)
        _visProc = VisionProcessor::getInstance();
    _sceneBBox = calcSceneCoords(_blobCluster->getBoundingBox());
    _blobCluster->assignVehicle(this);
    _name = name;
    _age = 0;
}

bool Vehicle::isValidVehicle(Blob *blob) {
    BoundingBox sc_box = calcSceneCoords(blob->getBoundingBox());
    double len = sc_box.length();
    double w = sc_box.width();
    if(len >= _MinVehicleLength && len <= _MaxVehicleLength
        && w >= _MinVehicleWidth && w <= _MaxVehicleWidth) {
        return true;
    }
    return false;
}

void Vehicle::show() {
    double *fcoord = _blobCluster->getBoundingBox().coordinates();
    long coord[4] = { fcoord[0], fcoord[1], fcoord[2], fcoord[3]};
    _visProc->showVehicle(coord, _type, _name);
//    Vector2d& vec = _blobCluster->getImgVelocity();
//    _visProc->showVelocity(vec.getX(), vec.getY());
}

void Vehicle::_replaceBlobs(BlobCluster *blobs)
{
    _blobCluster->replaceBlobs(blobs);
    _blobCluster->assignVehicle(this);
    _zombie = false;
    _age = 0;
}

void Vehicle::update() {
    double len_before = getSceneLength();
    double w_before = getSceneWidth();

    if(_zombie) {
        // should update velocity
        _age++;
        return;
    }

    _blobCluster->updateDimensions();
}

```

```

    _sceneBBox = calcSceneCoords(_blobCluster->getBoundingBox());

    double len_after = getSceneLength();
    double w_after = getSceneWidth();

    if(len_after <= 0.5*len_before || len_after < _MinVehicleLength)
    { //some of this vehicle's blobs have split
        _manager->findMissingBlobs(this);

#ifdef VEHICLE
        std::cout << "Vehicle " << _name << "requested blobs\n";
        std::cout << *_blobCluster << std::endl;
        cout << "Original length : " << len_before << " New length : " <<
len_after << endl;
#endif

    }

#ifdef VEHICLE
    std::cout << "Vehicle " << _name << " : " << *_blobCluster << std::endl;
#endif
}

void Vehicle::removeBlob(Blob *blob) {
    _blobCluster->removeBlob(blob);
    if(!_blobCluster->getNumBlobs())
        _zombie = true;
}

bool Vehicle::canMerge(BlobCluster& bclust) {
    double cg1[2], cg2[2];
    _sceneBBox.center(cg1);
    BoundingBox new_box = calcSceneCoords(bclust.getBoundingBox());
    new_box.center(cg2);
    double vertDist = fabs(cg1[1] - cg2[1]), horzDist = fabs(cg1[0] - cg2[0]);

    double length = _sceneBBox.length()/2 + new_box.length()/2+ horzDist;
    double width = _sceneBBox.width()/2 + new_box.width()/2+ vertDist;
    if((length <= _MaxVehicleLength) && (width <= _MaxVehicleWidth))
        return true;
    return false;
    //    &&
    //    (speedDiff <= 0.5*_imgVelocity.length()) && (theta <= 0.5))
}

```

```

BoundingBox Veh::calcSceneCoords(BoundingBox& img_box) {
    long imCoord[2];
    float scene_coord1[2], scene_coord2[2];
    Camera& _camera = Camera::getInstance();

    // imCoord[0] = img_box[0]; imCoord[1] = img_box[3];
    // _camera.getSceneCoords(imCoord, scene_coord1);
    imCoord[0] = img_box[0]; imCoord[1] = img_box[1];
    _camera.getSceneCoords(imCoord, scene_coord1);

    imCoord[0] = img_box[2]; imCoord[1] = img_box[3];
    _camera.getSceneCoords(imCoord, scene_coord2);

    BoundingBox scene_box(scene_coord1[0], scene_coord2[1],
                           scene_coord2[0], scene_coord1[1]);
    return scene_box;
}

```



```

// VehicleClassifier.cpp: implementation of the VehicleClassifier class.
//
/////////////////////////////////////////////////////////////////

#include "VehicleClassifier.h"

/////////////////////////////////////////////////////////////////
// Construction/Destruction
/////////////////////////////////////////////////////////////////

VehicleClassifier::VehicleClassifier() : _CarWidth(200)
{
}

VehicleClassifier::~~VehicleClassifier()
{
}

enum VehicleType VehicleClassifier::classify(Vehicle &veh) {
    float w = veh.getSceneWidth();
    if(w > _CarWidth)
        return Truck;
    else
        return Car;
    //check width, height, ratio of w:h or something and decide type
}

```

```

// VehicleMgr.cpp: implementation of the VehicleMgr class.
//
////////////////////////////////////////////////////////////////

#include "VehicleMgr.h"
#include "blobClusterer.h"
#include <iostream>

using std::list;
using std::cout;
using std::endl;

////////////////////////////////////////////////////////////////
// Construction/Destruction
////////////////////////////////////////////////////////////////

VehicleMgr::VehicleMgr() : _vehicleClassifier()
{
    Vehicle::setManager(this);
}

VehicleMgr::~VehicleMgr()
{
}

// Creates a vehicle only if the new (to be created) vehicle does not match any existing
ones
bool VehicleMgr::canCreateVehicle(Blob* blob)
{
    Vehicle *vh;
    if(Vehicle::isValidVehicle(blob)) {
        vh = new Vehicle(blob, _count++);
        if(!_isUniqueVehicle(vh)) {
            delete vh;
            _count--;
        }
        return true;
    }
    return false;
}

// Checks if the passed Vehicle is the same as some existing Vehicle (by means of spatial
matching)
bool VehicleMgr::_isUniqueVehicle(Vehicle *vh) {

```

```

    Vehicle *oldVeh;
    oldVeh = _isNewVehicle(vh);
    if(oldVeh) {
        oldVeh->_replaceBlobs(vh->_blobCluster);
        return false;
    }
    else {
        vh->_setType(_vehicleClassifier.classify(*vh));
        _vehicles.push_back(vh);
#ifdef VEHICLE_MGR
        cout << "New vehicle created. Length : " << vh-
>getSceneLength() << " Width : " << vh->getSceneWidth() << endl;
#endif

        return true;
    }
}

void VehicleMgr::showVehicles() {
    for(std::list<Vehicle*>::const_iterator itor = _vehicles.begin(); itor !=
_vehicles.end(); itor++) {
        (*itor)->show();
    }
}

void VehicleMgr::update() {
    for(std::list<Vehicle*>::iterator iter = _vehicles.begin(); iter != _vehicles.end();
iter++) {
        if((*iter)->_age > 2) {

#ifdef VEHICLE_MGR
            std::cout << "Vehicle " << (*iter)->_name << "age " << (*iter)-
>_age << " deleted" << std::endl;
#endif

            delete *iter;
            iter = _vehicles.erase(iter);

        }
        else {
            (*iter)->update();
            (*iter)->_setType(_vehicleClassifier.classify(**iter));
        }
    }
    if(_incompleteClusters.size()) {
        // remove all zombies

```

```

        list<BlobCluster*> clusters;
        for(std::list<Vehicle*>::iterator iter = _incompleteClusters.begin(); iter !=
_incompleteClusters.end(); iter++) {
            if((*iter)->_zombie)
                _incompleteClusters.erase(iter);
            else
                clusters.push_back((*iter)->_blobCluster);
        }
        _clusterer->findMatchingBlobs(clusters);
        _incompleteClusters.erase(_incompleteClusters.begin(),
_incompleteClusters.end());
    }
}

void VehicleMgr::findMissingBlobs(Vehicle *veh)
{
    _incompleteClusters.insert(_incompleteClusters.end(), veh);
}

Vehicle* VehicleMgr::_isNewVehicle(Vehicle *vh)
{
    float overlap, max_overlap=0;
    Vehicle *match=0;
    for(std::list<Vehicle*>::const_iterator iter = _vehicles.begin(); iter !=
_vehicles.end(); iter++) {
        overlap = (*iter)->_blobCluster->getBoundingBox().overlap(vh-
>_blobCluster->getBoundingBox());
        if(overlap > max_overlap) {
            max_overlap = overlap;
            match = *iter;
        }
    }
}

#ifdef VEHICLE_MGR
    std::cout << "Maximum overlap = " << max_overlap ;
    if(match)
        std::cout << "with vehicle " << match->_name ;
    std::cout << std::endl;
#endif

if(max_overlap >= .7)
    return match;

return 0;
}

```

```

// VisionProcessor.cpp: implementation of the VisionProcessor class.
//
////////////////////////////////////

#include "VisionProcessor.h"
#include "vehicle.h"

#include <iostream>

using std::cout;
using std::endl;
using std::list;

VisionProcessor* VisionProcessor::_instance = 0;

////////////////////////////////////
// Construction/Destruction
////////////////////////////////////

VisionProcessor* VisionProcessor::getInstance() {
    if(!_instance)
        _instance = new VisionProcessor();
    return _instance;
}

VisionProcessor::VisionProcessor() : _InitMinLength(10), _InitMaxLength(800),
    _BlobMinLength(50), _BlobMaxLength(700), _NumBuffers(3)
{
    /* Allocate the board, a thread and a camera */
    imDevAlloc(0, 0, NULL, IM_DEFAULT, &_device);
    imThrAlloc(_device, 0, &_thread);
    imCamAlloc(_thread, NULL, IM_DEFAULT, &_camera);

    /* Determine the image size */
    imCamInquire(_thread, _camera, IM_DIG_SIZE_X, &_sizeX);
    imCamInquire(_thread, _camera, IM_DIG_SIZE_Y, &_sizeY);
    // imDispInquire(_thread, 0, IM_DISP_RESOLUTION_X, &dispX);
    // imDispInquire(_thread, 0, IM_DISP_RESOLUTION_Y, &dispY);

    /* Allocate the display buffers at specific locations on the display */

    long screenBuff;
    imBufChild(_thread, IM_DISP, 0, 0, IM_ALL, IM_ALL, &screenBuff);
}

```

```

imBufClear(_thread, screenBuff, 0, 0);

imBufChild(_thread, IM_DISP, _sizeX+20, 50, _sizeX/2, _sizeY/2, &_resBuff);
imBufChild(_thread, IM_DISP, 0, 0, _sizeX/2, _sizeY/2, &_dispBuff[0]);
imBufChild(_thread, IM_DISP, _sizeX/2+10, 0, _sizeX/2, _sizeY/2,
&_dispBuff[1]);
imBufChild(_thread, IM_DISP, _sizeX/2+10, _sizeY/2+50, _sizeX/2, _sizeY/2,
&_blobBuff);

//Allocate the buffers for the grab buffer pool
for(int i = 0; i < _NumBuffers; i++) {
imBufAlloc2d(_thread, _sizeX/2, _sizeY/2, IM_UBYTE, IM_PROC,
&_grabPBuff[i]);
imSyncAlloc(_thread, &_grabOSB[i]);
}

imSyncAlloc(_thread, &_copyOSB);

//Allocate all the processing buffers
imBufAlloc2d(_thread, _sizeX/2, _sizeY/2, IM_UBYTE, IM_PROC,
&_edgePBuff);
imBufAlloc2d(_thread, _sizeX/2, _sizeY/2, IM_UBYTE, IM_PROC,
&_procPBuff);
imBufAlloc2d(_thread, _sizeX/2, _sizeY/2, IM_UBYTE, IM_PROC,
&_imPBuff[0]);
imBufAlloc2d(_thread, _sizeX/2, _sizeY/2, IM_UBYTE, IM_PROC,
&_imPBuff[1]);
imBufAlloc2d(_thread, _sizeX/2, _sizeY/2, IM_BINARY, IM_PROC,
&_idPBBuff);
imBufAlloc2d(_thread, _sizeX/2, _sizeY/2, IM_BINARY, IM_PROC,
&_edgePBBuff[0]);
imBufAlloc2d(_thread, _sizeX/2, _sizeY/2, IM_BINARY, IM_PROC,
&_edgePBBuff[1]);
imBufAlloc2d(_thread, _sizeX/2, _sizeY/2, IM_BINARY, IM_PROC,
&_procPBBuff);

imBufAlloc2d(_thread, _sizeX/2, _sizeY/2, IM_BINARY, IM_PROC,
&_idBBuff);
imBufAlloc2d(_thread, _sizeX/2, _sizeY/2, IM_BINARY, IM_PROC,
&_blobBBuff);

//Allocate and fill appropriate fields for the camera control buffer
imBufAllocControl(_thread, &_digControl);
imBufPutField(_thread, _digControl, IM_CTL_SUBSAMP_X, 2);
imBufPutField(_thread, _digControl, IM_CTL_SUBSAMP_Y, 2);

```

```

    imBufPutField(_thread, _digControl, IM_CTL_GRAB_MODE,
IM_ASYNCHRONOUS);

    //Allocate and fill up fields for the Warping function
    imBufAlloc2d(_thread, 3, 2, IM_FLOAT, IM_PROC, &_coeffBuff);
    imGenWarp1stOrder(_thread, _coeffBuff, IM_ROTATE, -9.0, 0, IM_CLEAR,
0);
    imGenWarp1stOrder(_thread, _coeffBuff, IM_TRANSLATE, 10.0, -40.0,
IM_NO_CLEAR, 0);
    imBufPutField(_thread, _coeffBuff, IM_CTL_RESAMPLE, IM_BILINEAR);
    imBufPutField(_thread, _coeffBuff, IM_CTL_OVERSCAN, IM_REPLACE);

    //Allocate buffer and select features for blob calculation
    imBlobAllocResult(_thread, &_result);
    imBlobAllocFeatureList(_thread, &_feature);
    imBlobSelectFeature(_thread, _feature, IM_BLOB_BOX, IM_DEFAULT);
    imBlobSelectFeature(_thread, _feature, IM_BLOB_LENGTH, IM_DEFAULT);
    imBlobSelectFeature(_thread, _feature, IM_BLOB_AREA, IM_DEFAULT);
    imBlobSelectFeature(_thread, _feature,
IM_BLOB_CENTER_OF_GRAVITY_X, IM_DEFAULT);
    imBlobSelectFeature(_thread, _feature,
IM_BLOB_CENTER_OF_GRAVITY_Y, IM_DEFAULT);

    //Control buffer for the erosion/dilation step
    imBufAllocControl(_thread, &_morphControl);
    imBufPutField(_thread, _morphControl, IM_CTL_OVERSCAN,
IM_REPLACE);

    //Control buffer for the graphics functions
    imBufAllocControl(_thread, &_graControl);
    imBufPutField(_thread, _graControl, IM_GRA_COLOR_MODE,
IM_PACKED);
    imBufPutField(_thread, _graControl, IM_GRA_COLOR, -1);

    //Control buffer for the text drawing functions
    imBufAllocControl(_thread, &_textControl);
    imBufPutField(_thread, _textControl, IM_GRA_FONT, IM_FONT_MEDIUM);

    _frames = _grabIndex = 0;
}

VisionProcessor::~VisionProcessor()
{
    //Free all allocated buffers;
    imBufFree(_thread, _resBuff);
    imBufFree(_thread, _dispBuff[0]);
}

```

```

    imBufFree(_thread, _dispBuff[1]);
    imBufFree(_thread, _blobBuff);

for(int i = 0; i < _NumBuffers; i++) {
    imBufFree(_thread, _grabPBuff[i]);
    imSyncFree(_thread, _grabOSB[i]);
}

imBufFree(_thread, _edgePBuff);
imBufFree(_thread, _procPBuff);
imBufFree(_thread, _imPBuff[0]);
imBufFree(_thread, _imPBuff[1]);
imBufFree(_thread, _idPBBuff);
imBufFree(_thread, _edgePBBuff[0]);
imBufFree(_thread, _edgePBBuff[1]);
imBufFree(_thread, _procPBBuff);
imBufFree(_thread, _idBBuff);
imBufFree(_thread, _blobBBuff);

imBufFree(_thread, _digControl);
imBufFree(_thread, _morphControl);
imBufFree(_thread, _graControl);
imBufFree(_thread, _textControl);

imBlobFree(_thread, _feature);
imBlobFree(_thread, _result);

imThrHalt(_thread, IM_FRAME);

/* Wait for everything to finish, then check for errors */
char Error[IM_ERR_SIZE]; // String to hold error message
imSyncHost(_thread, 0, IM_COMPLETED);
if (imAppGetError(IM_ERR_MSG_FUNC, Error))
    printf("%s\n", Error);

/* Clean up */
imCamFree(_thread, _camera);
imThrFree(_thread);
imDevFree(_device);
}

void VisionProcessor::initGrab () {
//    for(int i = 0; i < _NumBuffers-1; i++)
        imDigGrab(_thread, 0, _camera, _grabPBuff[0], 1, _digControl,
        _grabOSB[0]);

```



```

        imDigGrab(_thread, 0, _camera, _grabPBuff[1], 1, _digControl,
        _grabOSB[1]);
        imSyncHost(_thread, 0, IM_COMPLETED);

        imIntWarpPolynomial(_thread, _grabPBuff[0], _edgePBuff, _coeffBuff,
        _coeffBuff, 0);
        imIntConvolve(_thread, _edgePBuff, _imPBuff[0], IM_SMOOTH, 0, 0);
        imIntConvolve(_thread, _imPBuff[0], _procPBuff, IM_SOBEL_EDGE, 0, 0);
        imBinConvert(_thread, _procPBuff, _edgePBBuff[0],
        IM_GREATER_OR_EQUAL, 40, 0, 0);

        _grabIndex = 1 ; // _NumBuffers-2; // an additional -1 because there are
        0..._NumFrames-1 buffers
        _procIndex = 1;
    }

    int VisionProcessor::grabFrame() {
        imDigGrab(_thread, 0, _camera, _grabPBuff[_next(_grabIndex)], 1,
        _digControl, _grabOSB[_next(_grabIndex)]);

        imBufCopy(_thread, _grabPBuff[_prev(_grabIndex)], _dispBuff[0], 0, 0);
        imSyncHost(_thread, _grabOSB[_grabIndex], IM_COMPLETED);
        imBufCopy(_thread, _grabPBuff[_grabIndex], _dispBuff[1], 0, 0);
        int i = _grabIndex;
        _grabIndex = _next(_grabIndex);
        _frames++;
        return i;
    }

    void VisionProcessor::segmentImage(int frame_no) {
        imIntWarpPolynomial(_thread, _grabPBuff[frame_no], _edgePBuff, _coeffBuff,
        _coeffBuff, 0);
        imIntConvolve(_thread, _edgePBuff, _imPBuff[_procIndex], IM_SMOOTH, 0,
        0);
        imIntConvolve(_thread, _imPBuff[_procIndex], _procPBuff,
        IM_SOBEL_EDGE, 0, 0);
        imBinConvert(_thread, _procPBuff, _edgePBBuff[_procIndex],
        IM_GREATER_OR_EQUAL, 40, 0, 0);
        imBinTriadic(_thread, _edgePBBuff[_procIndex], _edgePBBuff[1-_procIndex],
        0, _idPBBuff, IM_PP_XOR, 0);
        _procIndex = 1 - _procIndex;

        imBufCopy(_thread, _edgePBuff, _resBuff, 0, _copyOSB);
    }

```

```

void VisionProcessor::calculateBlobs() {
    //binary dilation is faster, so convert to binary buffer
    imBinMorphic(_thread, _idPBBuff, _blobBBuff, IM_3X3_RECT_1,
IM_ERODE, 1, _morphControl, 0);
    imBinMorphic(_thread, _blobBBuff, _idPBBuff, IM_3X3_RECT_1,
IM_DILATE, 4, _morphControl, 0);

    //Now we have the final blobs, pass them through a size filter
    imBlobCalculate(_thread, _idPBBuff, 0, _feature, _result, IM_CLEAR, 0);
//    imBlobSelect(_thread, _result, IM_DELETE, IM_BLOB_LENGTH,
IM_DEFAULT, IM_OUT_RANGE, _BlobMinLength, _BlobMaxLength);
    imBufClear(_thread, _blobBuff, 0, 0);

    imBlobFill(_thread, _result, _blobBuff, IM_INCLUDED_BLOBS, -1, 0);
}

void VisionProcessor::showBlob(const blobData &bdata, const int name) {
    imGraRect(_thread, 0, _blobBuff, bdata.boundingBox[0], bdata.boundingBox[1],
        bdata.boundingBox[2], bdata.boundingBox[3]);

//    char t[5];
//    itoa(name, t, 10);
//    imGraText(_thread, _textControl, _blobBuff, bdata.centerGravity[0],
bdata.centerGravity[1], t);
}

list<blobData*>* VisionProcessor::getBlobs() {
    list<blobData*> *lst = new list<blobData*>;

    _count = imBlobGetNumber(_thread, _result, 0);
    imBlobGetResult(_thread, _result, IM_BLOB_BOX_X_MIN +
IM_TYPE_LONG, IM_DEFAULT, _minX);
    imBlobGetResult(_thread, _result, IM_BLOB_BOX_Y_MIN +
IM_TYPE_LONG, IM_DEFAULT, _minY);
    imBlobGetResult(_thread, _result, IM_BLOB_BOX_X_MAX +
IM_TYPE_LONG, IM_DEFAULT, _maxX);
    imBlobGetResult(_thread, _result, IM_BLOB_BOX_Y_MAX +
IM_TYPE_LONG, IM_DEFAULT, _maxY);
    imBlobGetResult(_thread, _result, IM_BLOB_AREA + IM_TYPE_LONG,
IM_DEFAULT, _area);
    imBlobGetResult(_thread, _result, IM_BLOB_CENTER_OF_GRAVITY_X +
IM_TYPE_LONG, IM_DEFAULT, _cgX);
    imBlobGetResult(_thread, _result, IM_BLOB_CENTER_OF_GRAVITY_Y +
IM_TYPE_LONG, IM_DEFAULT, _cgY);
}

```

```

        imBlobGetResult(_thread, _result, IM_BLOB_LABEL_VALUE +
IM_TYPE_LONG, IM_DEFAULT, _label);

        for(int i = 0; i < _count; i++) {
            blobData *bd = new blobData(_label[i], _minX[i], _minY[i], _maxX[i],
_maxY[i],
                _area[i], _cgX[i], _cgY[i]);
            lst->push_back(bd);
        }
        return lst;
    }

void VisionProcessor::showVehicle(long *bBox, enum VehicleType type, int name) {
    imGraRect(_thread, _graControl, _edgePBuff, bBox[0], bBox[1], bBox[2],
bBox[3]);
    imGraRect(_thread, _graControl, _edgePBuff, bBox[0]+1, bBox[1]+1, bBox[2]-
1, bBox[3]-1);
    // imGraRect(_thread, _graControl, _edgePBuff, bBox[0]+2, bBox[1]+2, bBox[2]-
2, bBox[3]-2);
    // char t[5];
    // itoa(name, t, 10);
    // imGraText(_thread, _textControl, VisionProcessor::resBuff, bBox[0]+(bBox[2] -
bBox[0])/2,
    //         bBox[1]+(bBox[3] - bBox[1])/2, t);
    if(type == Truck) {
        imGraLine(_thread, _graControl, _edgePBuff, bBox[0], bBox[1],
bBox[2], bBox[3]);
        imGraLine(_thread, _graControl, _edgePBuff, bBox[2], bBox[1],
bBox[0], bBox[3]);
        imGraLine(_thread, _graControl, _edgePBuff, bBox[0]-1, bBox[1]-1,
bBox[2]+1, bBox[3]+1);
        imGraLine(_thread, _graControl, _edgePBuff, bBox[2]+1, bBox[1]-1,
bBox[0]-1, bBox[3]+1);
    }
    imBufCopy(_thread, _edgePBuff, _resBuff, 0, _copyOSB);
    // imSyncHost(_thread, _copyOSB, IM_COMPLETED);
}

```

