THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

Department of Mechanical and Nuclear Engineering

Occlusion Detection in a Driving Scene By Registering Forward-looking Camera Images to a Rendered Map

Robert Leary
SPRING 2013

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Mechanical Engineering
with honors in Mechanical Engineering

Approved: _____ Date: _____

S. N. Brennan
Thesis Supervisor

_____ _____

H. J. Sommer III
Honors Adviser

Abstract

This thesis proposes a map-based vehicle occlusion detection algorithm by utilizing a vehicle's position on the road, maps of road features, and a forward-looking camera to find obstacles on the roadway. The challenge is that the mapped features must be rendered from the map database, registered to the image from the front of the vehicle, and then processed to determine occlusions. The use of a map simplifies the detection process by providing a known feature set of the road scene, where other algorithms must determine these features in real-time. The advantages of this approach include: confirmation of the vehicle's position relative to the road edge, updating of map information, rendering of map information through occluded scenes (to allow "see through" vehicles), optimization of image-processing thresholds as a function of map/position/lighting information, and provide knowledge of roads during harsh, low-visibility weather. This algorithm was successful in detecting occlusions of various road scenes, based on rendered maps of the environment.

Table of Contents

List of Figures

Acknowledgments

I would like to thank Dr. Brennan for giving me the opportunity to complete this research, as well as giving me direction on this thesis, even when I seemed so lost.

I want to thank Alex Brown for all of his help throughout the project, providing the vehicle data, and insight into ways to improve my work.

And most importantly, I want to thank my parents and my brothers. I owe *all* of my success to you.

Chapter 1

Introduction

The purpose of this thesis is to detect a vehicle's position on the road and obstacles on the roadway by matching maps of road features to images from a forward-looking camera. The challenge is that the mapped features must be rendered from the map database, registered to the image from the front of the vehicle, and then processed to determine occlusions. A flow-chart of the process is shown in Figure 1.1.

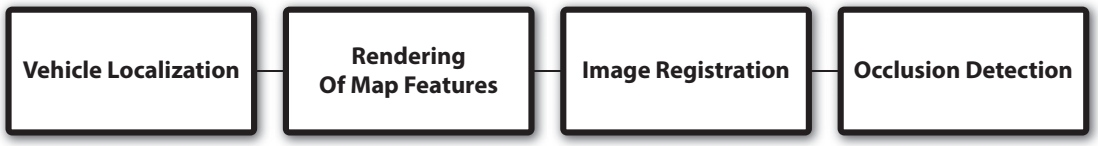

Figure 1.1: Flow-chart of the methods used for the occlusion detection process in this thesis.

The benefit of this approach, versus prior research on occlusion detection, is that the use of a map simplifies the detection process. Where most occlusion detection algorithms must determine the static environment through multiple image sequences [1, 2, 3, 4, 5, 6, 7], a map database provides these features quicker and more accurately due to the preprocessing of road scenes. Also, the use of one map database allows for multiple vehicles to share the same understanding of the local environment, as opposed to the potential errors in feature detection from a single

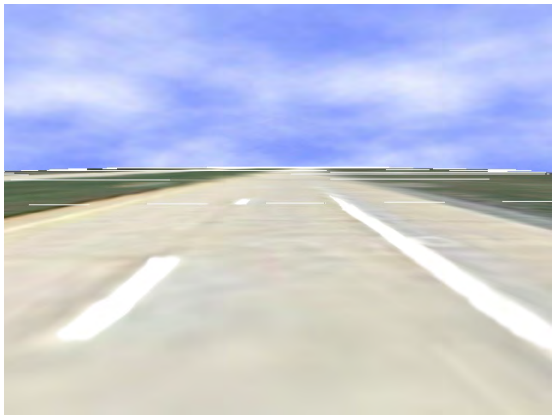

Figure 1.2: Gazebo rendering of camera view.

Figure 1.3: Vehicle forward-looking camera.

front-facing camera. These potential errors could be due to the mis-calibration, and perceived lighting and perspective of the road scene. Vehicle localization within the map provides insight into potential hazards ahead of the vehicle and provides a better understanding of the road for vehicle guidance, such as lane markers and turns within the road, prior to reaching those states.

An example of the proposed process is shown in Figure 1.2, which illustrates the rendered map image corresponding to the forward-looking camera image in Figure 1.3.

An advantage of this approach is that the occlusion detection process through map registration allows many benefits including: confirmation of the vehicle's position relative to the road edge, updating of map information, rendering of map information through occluded scenes (to allow "see through" vehicles), optimization of image-processing thresholds as a function of map, position, and lighting information, and provide knowledge of roads during harsh, low-visibility weather.

The remainder of the thesis is organized as follows:

Chapter 2, Literature Review, provides the history and current work within the fields of map-based vehicle informatics and occlusion detection. The methods for this thesis are outlined as well.

Chapter 3, Image Acquisition, outlines the equipment and equipment specifications used in this thesis. Also, the programming environments used for the image processing and map rendering are listed.

Chapter 4, Camera Calibration, specifically focuses on the underlying calculations of camera calibration including camera intrinsics, extrinsics, distortion coefficients and homographies.

Chapter 5, Map Construction, discusses the use of a three-dimensional rendering of the world, developed from Google Earth/Google SketchUp and rendered in Gazebo. Using collected vehicle data from a MEMS IMU and GPS, a virtual camera is localized within the rendered environment in the location of the real vehicle based on the east, north, up coordinates (ENU). This chapter also discusses the image registration methods used to register the map and camera views.

Chapter 6, Occlusion Detection, develops an algorithm for processing the registered map and camera images for occlusion detections. The disparity image represents the differences between what is expected in the world frame, and what actual is in the field of view of the camera. Of the remaining features, occlusions of sufficient size (width, length, area) are considered occlusions. This algorithm specifically focuses on vehicle occlusions, but can be extended to determine pedestrian occlusions as well.

Chapter 8, Conclusions, discusses the results of the occlusion detection process outlined in Figure 1.1, as well as future work to be completed to improve this research.

Chapter 2

Literature Review

This work combines map-based information retrieval, lane detection and image processing. All are mature areas of study, yet the combinations of these approaches remains relatively new. This chapter presents a summary of each research field as it applies to this thesis. Because each field has so much breadth, this thesis does not seek to comprehensively describe all research; rather, it focuses on only the most relevant contributions.

2.1 Map-based Vehicle Informatics

The implementation of map-based vehicle information systems have been developed to create a simulated driving environment. Past and current research has focused on driver interaction with a driving simulator, specifically to study driver decision making through behavioral studies [8, 9], as well as to assist in the rehabilitation of drivers to protect them from injuring themselves or others. [10] Also, the idea of providing a person with important information of the scene ahead, in the form of a heads-up display on the windshield, has been researched throughout the last 50 years [11]. These technologies are outlined below.

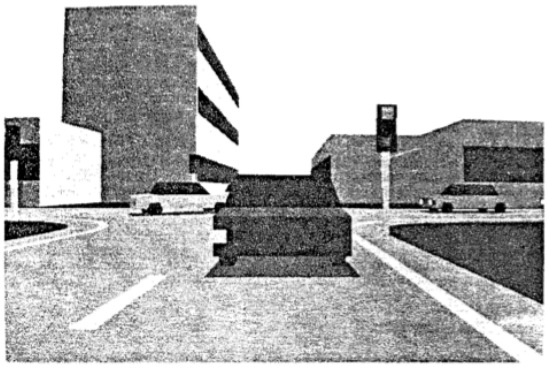

Figure 2.1: An example of a driving simulator developed in 1963. P. Kesling, *A New Driving Simulator Including An Interactive Intelligent Traffic Environment*. Accessed Apr. 07, 2013.

Figure 2.2: An example of a driving simulator developed in 2011. N. Fouladinejad, *Modeling virtual driving environment for a driving simulator*. Accessed Apr. 07, 2013.

2.1.1 Driving Simulators

A driving simulator is a virtual environment, focused on immersing an individual into a driving scenario that is modeled to closely resemble a real world driving scene. [12] This technology allows for the creation of low-cost simulation, where physical implementation may be expensive and dangerous. [13] In more recent work, the idea of a visual database of static terrain, such as roads and buildings, environmental features, such as foliage, and dynamic features, such as other motor vehicles has been researched to create a more realistic virtual environment. [12] Figure 2.2 and 2.1 display a simulated driving environment, one created in 1963 and one created in 2011.

2.1.2 Augmented Reality & Heads-up Displays

A heads-up display provides pertinent information to a driver, such as lane departure warnings [14] and driving directions [15]. This allows for messages of vehicle information to be projected onto a windshield, creating a form of augmented reality. Heads up displays have been heavily researched in the past for aircraft cockpits

[16], as well as motor vehicles [11], although generally only seeing implementations in military-based vehicles. [17] Modern motor vehicles are beginning to implement heads up displays for vehicle speed and driving directions[15].

2.2 Occlusion Detection

Past research in occlusion detection has focused on the segmentation of motion within a sequence of images to determine what is static, and what is dynamic within the images. This process is known as background modeling, or background subtraction. By removing stationary objects, such as roads, signs, and foliage, from an image, allows for the tracking of moving objects, or potential occlusions. [6, 7] Figure 2.3 shows and example of background modeling. These algorithms allowed for fast execution, given the limited amount of computational power at the time.

As computational speeds of computers have exponentially increased, more complex have been developed for image processing and occlusion detection that are still able to give real-time performance. More recent approaches utilize algorithms such template matching [18, 19, 20], three-dimensional modeling based on camera perspective, and state estimation algorithms [4, 21, 22], as outline below.

Most algorithms of current research still perform background detection with respect to a stationary camera based on the motion within the camera view [1, 2, 3, 4, 5]. Pixel motion is determined by the comparison of two consecutive image frames. Regions of minor or no pixel motion are assumed to be background features. Regions of high activity are considered for potential occlusion features. Background subtraction can be extended into the HSV-space for shadow elimination surrounding features of interest (Figure 2.4). [4]

Another approach in current research is template matching [18, 19, 20] based on an initial training set of multiple views of a range of vehicles and pedestrians.

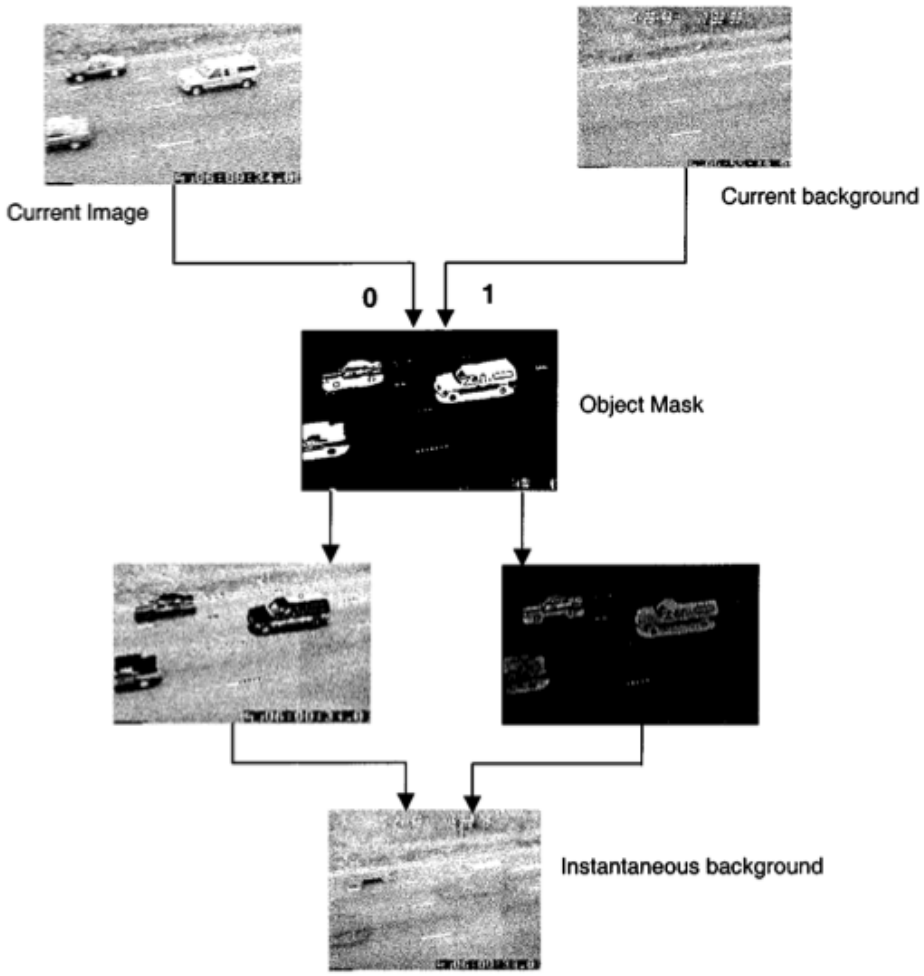

Figure 2.3: Example of background modeling. S. Gupte, O. Masoud, R. Martin, et al., *Detection and classification of vehicles*. Accessed Apr. 07, 2013.

Figure 2.5 shows an example of a training set used for vehicle detection. Each image captured is compared to the training set to determine the likelihood of a match. Template matching is also used in road sign detection (Figure 2.6). [23]

Features can be tracked throughout image sequences using state estimation algorithms such as a Kalman filter (Figure 2.7) [4] or a Particle filter [21]. Also, the Lucas Kanade algorithm has been used for feature tracking based on optical flow [22]. These algorithms improve the accuracy of occlusion detection as features are tracked throughout their entire motion, instead of each frame being considered

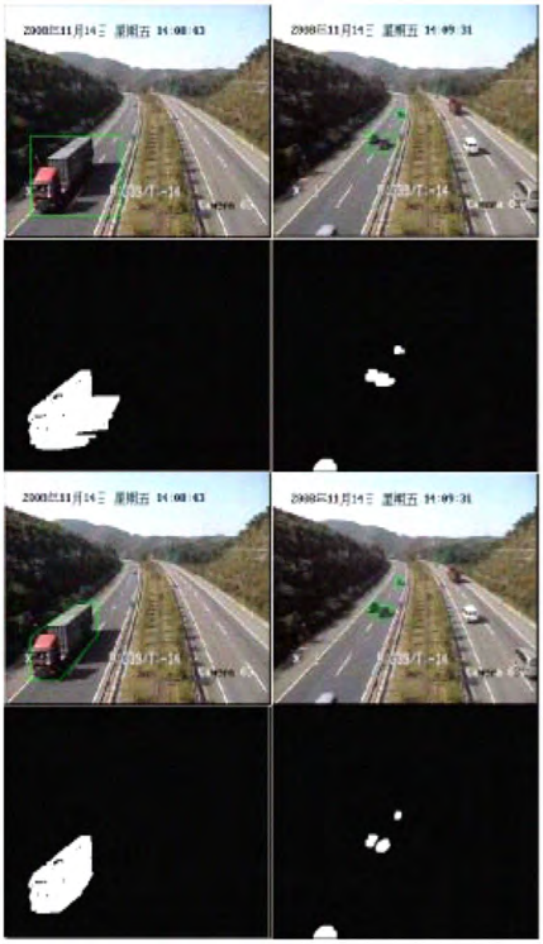

Figure 2.4: Example of shadow elimination. J. Luo and J. Zhu, *Improved Video-Based Vehicle Detection Methodology*. Accessed Mar. 22, 2013.

a new scenario.

Once the occlusions are identified, it can be beneficial to determine a three-dimensional bounding box to represent the footprint of the occlusion in the world space. Three-dimensional deformable models can be developed based on the expected size of the occlusion and thresholded based on the dimensions of the bounding box. [5]

Figure 2.5: Example of training set used for vehicle detection. C. Wang, and J. Lien, *Features A Statistical Approach*. Accessed Mar. 27, 2013.

Figure 2.6: Example of training set used for sign detection. S. Maldonado-bascon, S. Lafuente-arroyo, P. Gil-jimenez et al., *Road-Sign Detection and Recognition Based on Support Vector Machines*. Accessed Mar. 20, 2013.

2.3 Outline of Methods Used

In this thesis, the application of a map-based simulated environment is used to map road features to a forward-facing camera. The map database provides important information of the surrounding environment and can be used as a reference for the feature comparisons of the map and forward-facing image features. The use of basic image processing filters, such as morphological and binary filters, are used for occlusion detection. These filters are computationally efficient, easy to implement, and provide satisfactory results, as outlined above from past research in occlusion detection. Occlusion contours are thresholded based on the width, height,

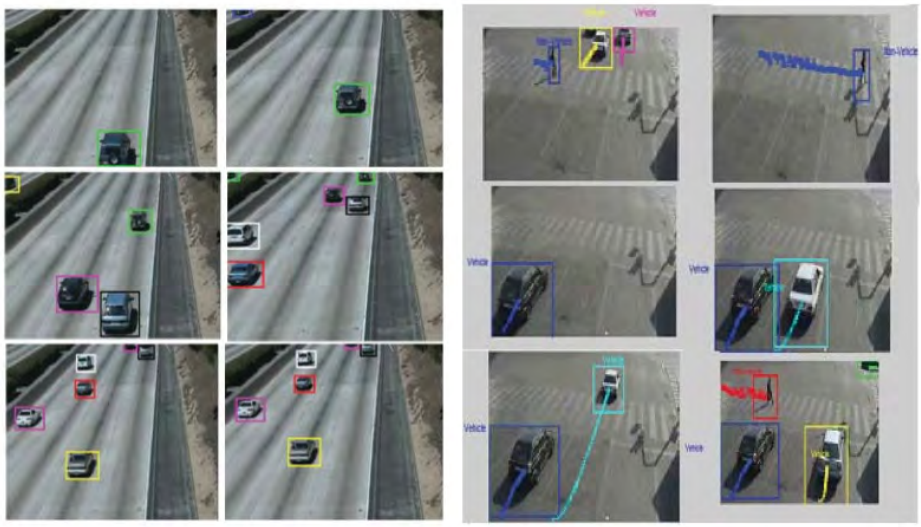

Figure 2.7: Example of the use of a Kalman filter to track occlusions. A. Ghasemi, R. Safabakhsh, *A real-time multiple vehicle classification and tracking system with occlusion handling*. Accessed Mar. 15, 2013.

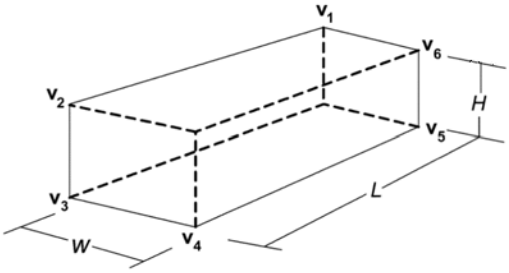

Figure 2.8: Example of deformable three-dimensional model. C. Pang, *A novel method for resolving vehicle occlusion in a monocular traffic-image sequence*. Accessed Mar. 15, 2013.

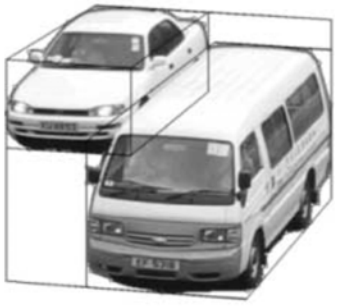

Figure 2.9: Example of deformable three-dimensional model fit to road scene. C. Pang, *A novel method for resolving vehicle occlusion in a monocular traffic-image sequence*. Accessed Mar. 15, 2013.

and area. This approach simplifies the analysis of potential occlusions, instead of developing a three-dimensional model of occlusions based on the occlusion position and perspective of the forward-facing camera, as discussed above. These are the methods used within this thesis.

Chapter 3

Image Acquisition

3.1 Camera & Computer

For algorithm verification, testing was completed with an Apple iSight webcam on a 2008 MacBook Pro and images taken with an iPhone. Video was obtained using a Point Grey Firefly MV Mono USB camera (Figure 3.1) with a wide-angle lens. Images were acquired at the default 15 frames per second at a resolution of 640x480. All testing was completed on a 2008 MacBook Pro 2.53 GHz Intel Core 2 Duo laptop with 4 GB RAM.

Figure 3.1: Point Grey Firefly MV Mono USB camera. ptgrey.com. Accessed Apr. 03, 2013.

3.2 Programming Environment

All algorithms were implemented and tested in Python (v. 2.7.3) using the OpenCV library (v. 2.4.3). All maps were constructed through Google Earth/SketchUp and implemented through the Robot Operating System (ROS) and Gazebo.

3.3 Image Collection Environments

North Atherton Street Bridge

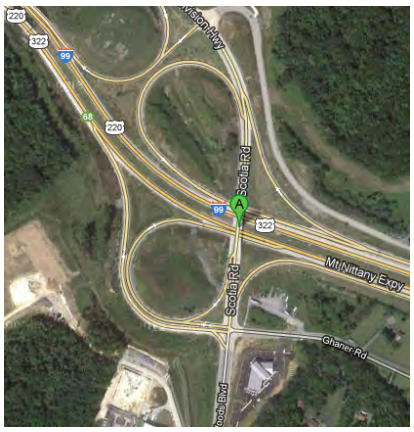

Figure 3.2: Location of out-of-vehicle image acquisition. Images were taken from North Atherton Street Bridge in central Pennsylvania.

For initial algorithm testing, out-of-vehicle images were obtained from North Atherton Street Bridge (Figure 3.2) in central Pennsylvania. Images were taken on an overcast day in March 2013. This location was chosen because of the distinct lane markers, abundance of multi-vehicle scenes, and unobtrusiveness for image acquisition. The algorithms discussed in this thesis processed the images off-line for initial algorithm verification.

Route 322 North

In-vehicle video, GPS, and MEMS IMU data was obtained on Route 322 North (Figure 3.3) in central Pennsylvania on a sunny day in March 2013. This location was chosen for the sparse, basic highway scenes for video and image based algorithm

Figure 3.3: Location of in-vehicle video/data acquisition. Video and data was collected on Route 322 North in central Pennsylvania.

testing. All videos and images were processed off-line, as real-time performance could not be achieved. In-vehicle video was chosen because a goal of the thesis is to develop an in-vehicle occlusion detection system.

Chapter 4

Camera Calibration

Camera calibration allows for images to be undistorted and real world distances to be determined. The approach explained in this chapter utilizes homogeneous coordinates to determine the camera calibration matrix, distortion coefficients and homography matrix.

4.1 Pinhole Projection with Homogeneous Coordinates

The transformation of 3D world points to 2D image points is non-linear, as many points in the world space correspond to the same pixel on the imaging sensor. Therefore, the use of homogeneous coordinates simplifies the problem to a linear transformation.

$$\mathbf{W} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \Rightarrow \mathbf{W} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} \tag{4.1}$$

Any non-zero scalar multiple of a homogeneous vector pertains to the same

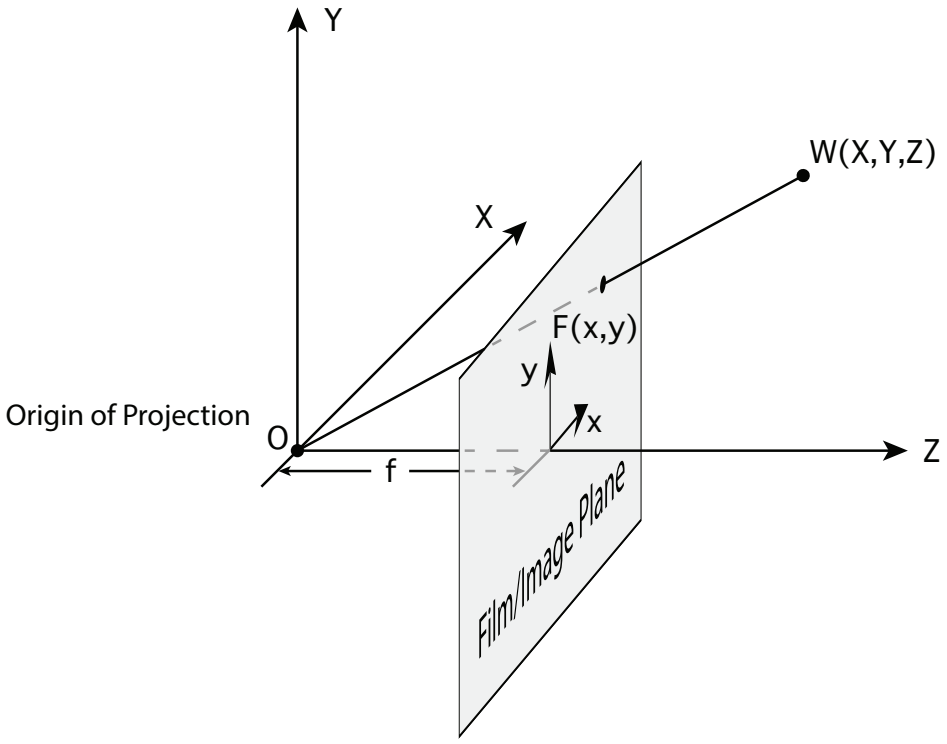

Figure 4.1: Coordinate systems used to model a pinhole projection. This shows the relation of the origin of projection to an arbitrary point in the world space.

image point.

4.2 Intrinsic Parameters

From Figure 4.1, the relationship between a point on the Film Plane, $F(x, y)$, to a point in the World Space, $W(X, Y, Z)$, can be determined through similar triangles. This relationship assumes a common center of projection, labeled as O in Figure 4.1. The parameter f is known as the focal length, and is an inherent property of the camera. [24, 25]

$$\frac{f}{Z} = \frac{x}{X} = \frac{y}{Y} \tag{4.2}$$

Solving 4.2 for x and y, the location of the three-dimensional point on the film

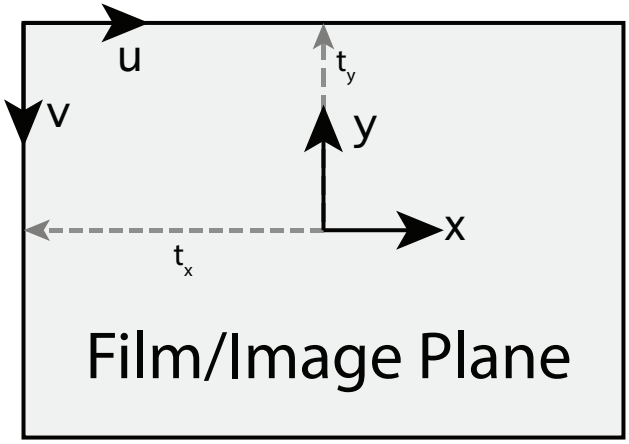

Figure 4.2: Camera sensors present data starting from the top left of the Image Plane.

plane can be determined:

$$x = \frac{fX}{Z} \tag{4.3}$$

$$y = \frac{fY}{Z} \tag{4.4}$$

In matrix notation:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{4.5}$$

The conversion from the non-linear matrix to a homogeneous matrix is as follows:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} \tag{4.6}$$

Most commercial CCD/CMOS imaging sensors do not consider the center of the image plane as the origin of the image plane, but the top-left of the image. Therefore, $Image(0, 0)$ starts in the top-left, and $Image(width, height)$ ends at the

bottom-right of the image. This transformation introduces the translation from the center of the image to the top-left.

$$u' = \frac{fX}{Z} + t_x \tag{4.7}$$

$$v' = \frac{fY}{Z} + t_y \tag{4.8}$$

In homogeneous matrix notation:

$$\begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} = \begin{bmatrix} f & 0 & t_x & 0 \\ 0 & f & t_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} \tag{4.9}$$

If the pixels of the CCD/CMOS sensor are not square, a scaling factor can be introduced for each axis of the image.

$$u' = m_x \frac{fX}{Z} + m_x t_x \tag{4.10}$$

$$v' = m_y \frac{fY}{Z} + m_y t_y \tag{4.11}$$

Note: m_x/m_y is known as the aspect ratio.

In homogeneous matrix notation:

$$\begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} = \begin{bmatrix} m_x f & 0 & m_x t_x & 0 \\ 0 & m_y f & m_y t_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} \tag{4.12}$$

If the pixel axes are not orthogonal, a skew parameter, s, can be introduced.

$$\begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} = \begin{bmatrix} m_x f & s & m_x t_x & 0 \\ 0 & m_y f & m_y t_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} \tag{4.13}$$

Note: s is usually close to zero.

Equation 4.13 can be simplified to the following matrix multiplication.

$$\begin{bmatrix} u' \\ v' \\ w' \end{bmatrix} = \begin{bmatrix} \beta_x & s & u_o \\ 0 & \beta_y & v_o \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = \mathbf{K} \begin{bmatrix} \mathbf{I}_3 & \mathbf{0}_3 \end{bmatrix} \mathbf{P} \tag{4.14}$$

Matrix $\mathbf{K}$, is known as the *intrinsic* parameter matrix. This matrix is an upper triangular matrix and has five degrees of freedom.

The actual pixel values are determined from the following ratios:

$$u = \frac{u'}{w'} \tag{4.15}$$

$$v = \frac{v'}{w'} \tag{4.16}$$

4.3 Extrinsic Parameters

While the intrinsic parameters are inherent to the imaging sensor, the extrinsic parameters are the transformation from the world frame to the camera frame. The transformation between these frames is a rotation and translation about the three coordinate axes.

The rotation from the world frame to the camera frame is a matrix multiplication of the individual rotations about the corresponding axes. [25]

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\theta & -sin\theta \\ 0 & sin\theta & cos\theta \end{bmatrix} \tag{4.17}$$

$$\mathbf{R}_y(\alpha) = \begin{bmatrix} cos\alpha & 0 & sin\alpha \\ 0 & 1 & 0 \\ -sin\alpha & 0 & cos\alpha \end{bmatrix} \tag{4.18}$$

$$\mathbf{R}_z(\phi) = \begin{bmatrix} cos\phi & -sin\phi & 0 \\ sin\phi & cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.19}$$

$$\mathbf{R} = \mathbf{R}_x\mathbf{R}_y\mathbf{R}_z \tag{4.20}$$

$$\mathbf{T} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \tag{4.21}$$

The combination of the rotation and translation matrices results in the *extrinsic* parameter matrix, **E**:

$$\mathbf{E} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{T} \\ \mathbf{0}_3^T & 1 \end{bmatrix} \tag{4.22}$$

4.4 Camera Calibration

In real world application, it can be difficult to accurately model the camera intrinsic and extrinsic parameters, therefore it is easier to take point correspondents

from the world frame and the image frame and solve for the transformation matrices. The conversion from world to image coordinates can modeled with a linear transformation with the use of homogeneous coordinates. [25]

$$\mathbf{x}_i = \mathbf{P}\mathbf{X}_i \tag{4.23}$$

where,

$$\mathbf{x}_i = \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{4.24}$$

$$\mathbf{X}_i = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} \tag{4.25}$$

With homogeneous coordinates, $\mathbf{x_i}$ and $\mathbf{P}\mathbf{X}_i$ must be parallel.

$$\mathbf{x}_i \times \mathbf{P}\mathbf{X}_i = 0 \tag{4.26}$$

For ease of calculation, let $\mathbf{p}_1^T$, $\mathbf{p}_2^T$, $\mathbf{p}_3^T$ be the rows of $\mathbf{P}$.

$$\mathbf{P}\mathbf{X}_i = \begin{bmatrix} \mathbf{p}_1^T X_i \\ \mathbf{p}_2^T X_i \\ \mathbf{p}_3^T X_i \end{bmatrix} \tag{4.27}$$

$$\mathbf{x}_i \times \mathbf{P}\mathbf{X}_i = \begin{bmatrix} v_i \mathbf{p}_3^T \mathbf{X}_i - w_i \mathbf{p}_2^T \mathbf{X}_i \\ w_i \mathbf{p}_1^T \mathbf{X}_i - u_i \mathbf{p}_3^T \mathbf{X}_i \\ u_i \mathbf{p}_2^T \mathbf{X}_i - v_i \mathbf{p}_1^T \mathbf{X}_i \end{bmatrix} \tag{4.28}$$

Breaking Equation 4.28 into matrix notation, the unknown vectors $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$

can be solved.

$$\begin{bmatrix} \mathbf{0}_4^T & -w_i\mathbf{X}_i^T & v_i\mathbf{X}_i^T \\ w_i\mathbf{X}_i^T & \mathbf{0}_4^T & -u_i\mathbf{X}_i^T \\ -v_i\mathbf{X}_i^T & u_i\mathbf{X}_i^T & \mathbf{0}_4^T \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \mathbf{p}_2 \\ \mathbf{p}_3 \end{bmatrix} = \mathbf{0} \tag{4.29}$$

Note: $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$ creates a 12x1 vector.

The matrix in Equation 4.28 is rank two, with 12 unknowns. If scale is ignored (the last entry of $\mathbf{P}$ is forced to be 1), there are 11 unknowns. Matrix $\mathbf{P}$ is susceptible to noise error; therefore multiple point correspondents help to reduce the noise error. With the use of a chessboard for calibration, each corner of each square is one point, so using a fairly large chessboard will help mitigate the error, but more importantly, different viewing angles of the same chessboard, with many squares will more accurately approximate the transformation matrix, $\mathbf{P}$. Each new point correspondence is appended to the matrix in Equation 4.28. Singular Value Decomposition (SVD) can be used to solve Equation 4.28, with the optimal solution for matrix $\mathbf{P}$ corresponding to the last column of $\mathbf{V}$ and the smallest singular value.

4.5 Camera Translation

The translation of the camera ($\mathbf{C}$), from world to camera coordinates, can be determined from the null space of matrix $\mathbf{P}$.

$$\mathbf{PC} = \mathbf{0} \tag{4.30}$$

Matrix $\mathbf{P}$ is a 3x4 matrix; therefore the solution to Equation 4.30 can be determined through SVD of $\mathbf{P}$. The translation vector $\mathbf{C}$ is the last column of $\mathbf{V}$, corresponding to the smallest singular value.

4.6 Camera Pose and Intrinsic Parameters

The camera pose and intrinsic parameters can be determined through RQ decomposition of the left 3x3 sub matrix ($\mathbf{M}$) of $\mathbf{P}$. RQ decomposition is similar to the standard QR decomposition. QR decomposition forms an orthogonal matrix $\mathbf{Q}$ from the column space of $\mathbf{P}$, beginning with the first column of $\mathbf{P}$. RQ decomposition forms an orthogonal matrix formed from the row space of $\mathbf{P}$, beginning with the last row of $\mathbf{P}$.

$$\mathbf{M} = \begin{bmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{bmatrix} \tag{4.31}$$

Let $\mathbf{m}_1$, $\mathbf{m}_2$, $\mathbf{m}_3$ be the rows of $\mathbf{M}$. For RQ decomposition, matrix M must be rotated 180° and transposed.

$$\mathbf{M}' = \begin{bmatrix} \vdots & \vdots & \vdots \\ \mathbf{m}_3 & \mathbf{m}_2 & \mathbf{m}_1 \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{4.32}$$

Matrix $\mathbf{M}'$ can be now be decomposed into $\mathbf{Q}$ and $\mathbf{R}$ using the Gram-Schmidt method. Matrix $\mathbf{Q}$ is the intrinsic parameters matrix $\mathbf{K}$ described in (Section 4.2), and matrix $\mathbf{R}$ is the rotational matrix $\mathbf{R}$ described in (Section 4.3).

4.7 Lens Distortion Model

Low-end cameras are susceptible to two types of distortion: radial and tangential distortion. Radial distortion, also known as barrel or fish eye distortion, occurs when the lens magnification is unequally distributed axially from the center of the image. Tangential distortion occurs when the lens is not parallel to the imaging sensor plane (CCD/CMOS sensor). [26]

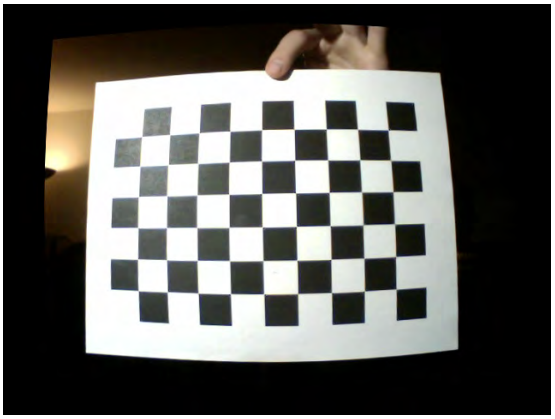

Figure 4.3: Uncalibrated image.

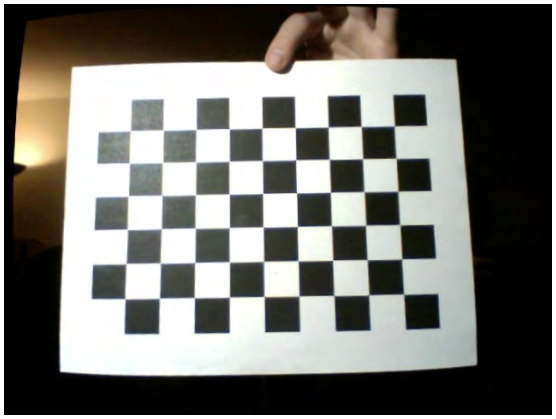

Figure 4.4: Calibrated

$$x_c - x_o = L(r)(x - x_o) \tag{4.33}$$

$$y_c - y_o = L(r)(y - y_o) \tag{4.34}$$

$$r^2 = (x - x_o)^2 + (y - y_o)^2 \tag{4.35}$$

$$f(\kappa_1, \kappa_2) = \sum_i (x_i - x_{ci})^2 + (y_i - y_{ci})^2 \tag{4.36}$$

$$L(r) = 1 + \kappa_1 r^2 + \kappa_2 r^4 + \ldots \tag{4.37}$$

Radial Distortion Equations:

$$x_{corrected} = x(1 + \kappa_1 r^2 + \kappa_2 r^4 + \ldots) \tag{4.38}$$

$$y_{corrected} = y(1 + \kappa_1 r^2 + \kappa_2 r^4 + \ldots) \tag{4.39}$$

Tangential Distortion Equations:

$$x_{corrected} = x + [2t_1xy + t_2(r^2 + 2x^2)] \tag{4.40}$$

$$y_{corrected} = y + [2t_2xy + t_1(r^2 + 2y^2)] \tag{4.41}$$

The final distortion equations can be determined by combining the radial and tangential distortion equations:

$$\delta(x) = x(1 + \kappa_1r^2 + \kappa_2r^4) + [2t_1xy + t_2(r^2 + 2x^2)] \tag{4.42}$$

$$\delta(y) = y(1 + \kappa_1r^2 + \kappa_2r^4) + [2t_2xy + t_1(r^2 + 2y^2)] \tag{4.43}$$

4.8 Planar Homography

The calibration process explained thus far allows for the projection of world points to image points. The reverse, the projection of image points to world coordinates, is impossible, as the matrix **P** is 3x4 and subsequently non-invertible. One way to circumvent this issue is to assume a planar projection: the world space reduced to a two-dimensional space from a three-dimensional space. An example of this to assume the ground is flat, and therefore the z-direction of the world space is zero. This allows for measurements to be made within the ground plane, but object height is unattainable. [27]

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} X \\ Y \\ 0 \\ W \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{14} \\ p_{21} & p_{22} & p_{24} \\ p_{31} & p_{32} & p_{34} \end{bmatrix} \begin{bmatrix} X \\ Y \\ W \end{bmatrix} \tag{4.44}$$

The new matrix is known as the homography matrix, **H**.

$$\mathbf{H} = \begin{bmatrix} p_{11} & p_{12} & p_{14} \\ p_{21} & p_{22} & p_{24} \\ p_{31} & p_{32} & p_{34} \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \tag{4.45}$$

Note: Matrix $\mathbf{H}$ is invertible.

To convert from world coordinates to image coordinates:

$$\mathbf{p} = \mathbf{HX} \tag{4.46}$$

To convert from image coordinates to world coordinates.

$$\mathbf{X} = \mathbf{H}^{-1}\mathbf{p} \tag{4.47}$$

In some circumstances it may be beneficial to impose the extrinsic parameters by hand and not through the iterative calculation of matrix $\mathbf{P}$. For example, a camera mounted on the windshield of a vehicle has a measurable pose and vertical translation from the ground plane. This approach requires prior knowledge of the camera intrinsic matrix $\mathbf{K}$. Matrix $\mathbf{P}$ can be calculated from the following matrix multiplication:

$$\mathbf{P} = \mathbf{KE} \tag{4.48}$$

where, as explained in Section 4.3, $\mathbf{E}$ is the extrinsic parameters matrix. Matrix $\mathbf{P}$ can be used to project 3D world points to the 2D image plane. Figure 4.5 depicts a simple application of an imposed translation between the camera and the world frame ($\mathbf{t}$), and a specified point in the world frame ($\mathbf{w}$) to be projected.

The calculation of the new homography is the same process explained earlier in this section, where the ground plane (Z) is considered zero, and the system reduces to a 3x3 invertible matrix $\mathbf{H}$.

Camera calibration and homography code can be found in Appendix A.2.

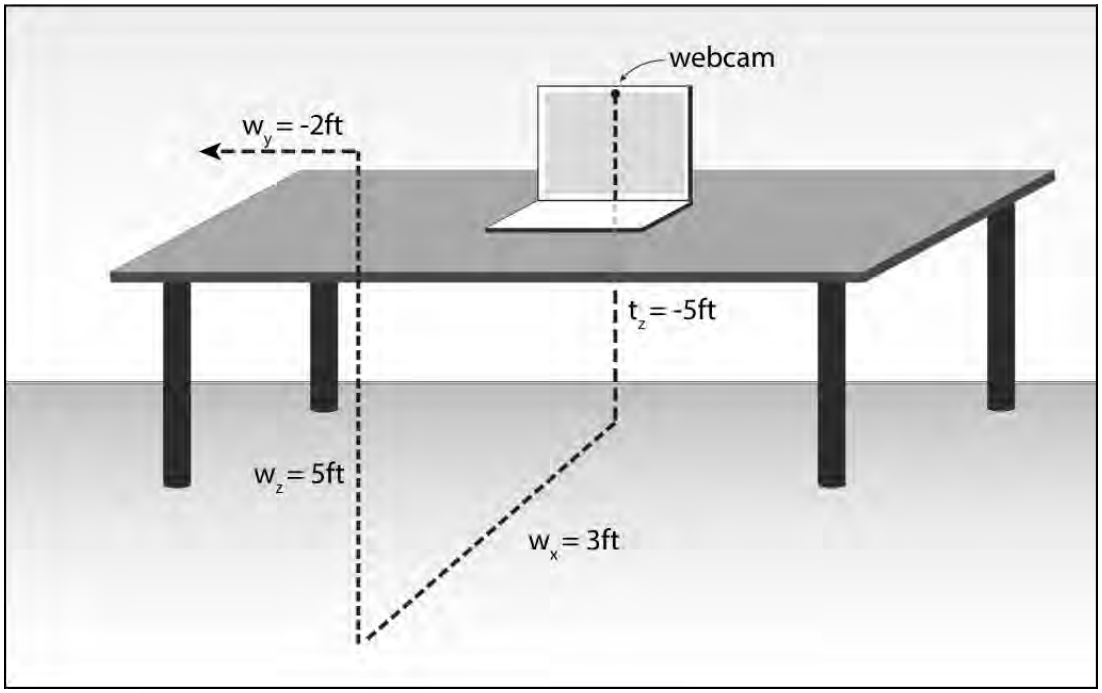

Figure 4.5: An application of an imposed translation, $\mathbf{t} = [0, 0, -5]^T$, between the camera world frame, and a 3D point, $\mathbf{w} = [3, -2, 5]^T$.

Chapter 5

Map Construction

In this chapter, a procedure for developing a three-dimensional map rendering of a local environment is outlined. Using vehicle data collected from a GPS and MEMS IMU, a program is developed to position a virtual vehicle within the rendered environment at the exact position as in the real world. By localizing a virtual vehicle within the rendered environment, an approximate view of what should and should not be in front of the vehicle can be determined. (See Figures 5.1 & 5.2) In order to compare the two images, they must first be registered to each other. The Speeded Up Robust Features and Lucas Kanade algorithms are implemented for the image registration. The Lucas Kanade algorithm was used in the final implementation of this thesis.

5.1 Data Acquisition

In order to localize the vehicle both within the world and the map, an accurate position and orientation of the vehicle must be determined. Data was collected from Route 322 using a MEMS inertia measurement unit (IMU) for vehicle velocity and orientation, a global positioning system (GPS), and a Point Grey Firefly MV Mono USB camera for image acquisition. The GPS coordinates (WGS84) were

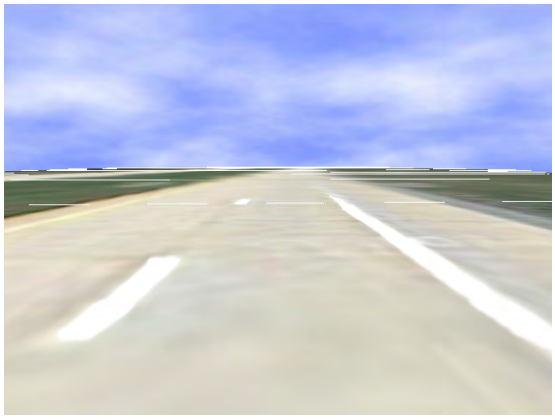

Figure 5.1: Gazebo rendering of camera view.

Figure 5.2: Vehicle camera view image.

converted from latitude (ϕ), longitude (λ), and altitude (h) to local east (x), north (y), up (z) coordinates (ENU) to position the vehicle within a local coordinate system, instead of a global coordinate system. The WGS84 coordinates must first be converted to the earth centered, earth fixed (ECEF) coordinate frame using the following equations and constants: [28]

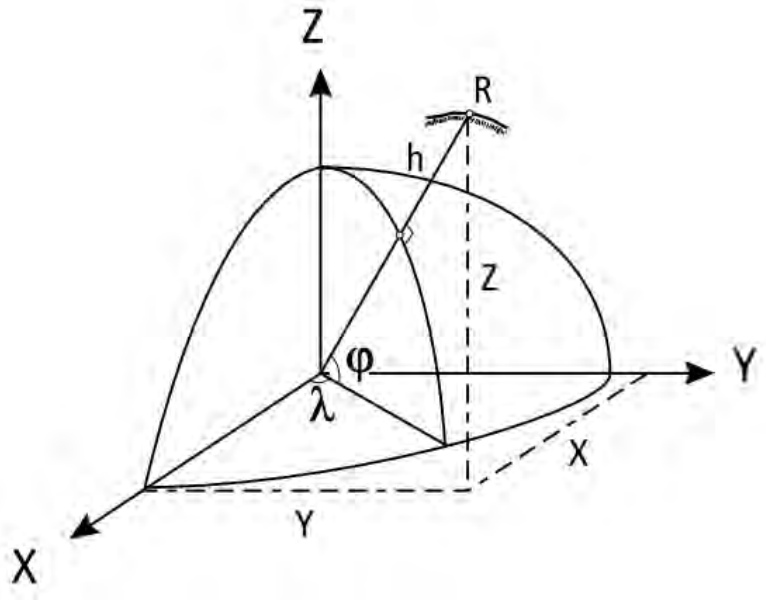

Figure 5.3: Coordinate conversion from WGS84 to Earth Centered, Earth Fixed. NAL Research. http://www.nalresearch.com. Accessed Mar. 29, 2013.

$$a = 6378137m$$

$$f = 1/298.257223563$$

$$b = a(1 - f) \tag{5.1}$$

$$e = \sqrt{\frac{a^2 - b^2}{a^2}} \tag{5.2}$$

$$\begin{aligned} X &= (N + h)cos\phi cos\lambda \\ Y &= (N + h)cos\phi sin\lambda \\ Z &= (\frac{b^2}{a^2}N + h)sin\phi \end{aligned} \tag{5.3}$$

where, N is the radius of curvature of the earth, defined as:

$$N = \frac{a}{\sqrt{1 - e^2 sin^2\phi}} \tag{5.4}$$

After the coordinates are transformed to ECEF coordinates, they must be transformed to local ENU coordinates using the following equation:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -sin\lambda_{ref} & cos\lambda_{ref} & 0 \\ -cos\lambda_{ref}sin\phi_{ref} & -sin\lambda_{ref}sin\phi_{ref} & cos\phi_{ref} \\ cos\lambda_{ref}cos\phi_{ref} & sin\lambda_{ref}cos\phi_{ref} & sin\phi_{ref} \end{bmatrix} \begin{bmatrix} X - X_{ref} \\ Y - Y_{ref} \\ Z - Z_{ref} \end{bmatrix} \tag{5.5}$$

where X_{ref}, Y_{ref}, and Z_{ref} is a reference point in the ECEF coordinate frame for the origin of the ENU map and ϕ_{ref} and λ_{ref} is the corresponding WSG84 coordinates of the reference point.

Example calculation of local ENU coordinates from WSG84 coordinates:

Reference latitude, longitude, and altitude:

$$\phi_{ref} = 40.79115482216408$$
$$\lambda_{ref} = -78.06739040855355$$
$$h_{ref} = 328.5669250488281$$

Desired latitude, longitude, and altitude location:

$$\phi = 40.79035174276196$$
$$\lambda = -78.06735719758326$$
$$h = 321.7445678710938$$

Note: All latitude, longitude, and altitude coordinates must be converted from degrees to radians.

Determine the earth centered, earth fixed coordinate:

$$b = a(1 - f) = 6356752.31m$$
$$e = \sqrt{\frac{a^2 - b^2}{a^2}} = \sqrt{\frac{(6378137m)^2 - (6356752m)^2}{(6378137m)^2}} = 0.08181919084$$

Reference coordinate conversion to ECEF:

$$N_{ref} = \frac{a}{\sqrt{1 - e^2 sin^2 \phi_{ref}}} = 6387268.38m$$

$$X_{ref} = (N_{ref} + h_{ref})cos\phi_{ref}cos\lambda_{ref} = 999850.85$$
$$Y_{ref} = (N_{ref} + h_{ref})cos\phi_{ref}sin\lambda_{ref} = -4731285.07$$
$$Z_{ref} = (\frac{b^2}{a^2}N_{ref} + h)sin\phi_{ref} = 4144895.53$$

Desired coordinate conversion to ECEF:

$$N = \frac{a}{\sqrt{1 - e^2 sin^2\phi}} = 6387268.08m$$

$$X = (N + h)cos\phi cos\lambda = 999865.62m$$
$$Y = (N + h)cos\phi sin\lambda = -4731341.41m$$
$$Z = (\frac{b^2}{a^2}N + h)sin\phi = 4144827.94m$$

Determine the local east, north, up coordinate:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -sin\lambda_{ref} & cos\lambda_{ref} & 0 \\ -cos\lambda_{ref}sin\phi_{ref} & -sin\lambda_{ref}sin\phi_{ref} & cos\phi_{ref} \\ cos\lambda_{ref}cos\phi_{ref} & sin\lambda_{ref}cos\phi_{ref} & sin\phi_{ref} \end{bmatrix} \begin{bmatrix} X - X_{ref} \\ Y - Y_{ref} \\ Z - Z_{ref} \end{bmatrix} = \begin{bmatrix} 11.479m \\ 17.116m \\ -86.813m \end{bmatrix}$$

The vector containing x, y, and z represents the local east, north, and up coordinates relative to the reference coordinate.

5.2 ROS & Gazebo

Robot Operating System (ROS) provides an operating system for the simulation and development of robotic control. For testing, vehicle data was recorded on

Route 322 in Pennsylvania using the sensors outline in Section 5.1. ROS was used to process the images for the occlusion detection algorithm, as well as process vehicle positioning data from the MEMS IMU and GPS. The ENU coordinates calculated from the equations in Section 5.1 were passed through to Gazebo to position a virtual camera within a rendered map environment.

Gazebo is a three-dimensional robot simulator, capable of simulating various sensors and robot models with rigid-body physics. [29] A camera sensor was used within Gazebo to virtually simulate a camera mounted on a vehicle. Figure 5.1 shows the Gazebo rendering of the environment from the perspective of a virtual camera mounted on a virtual vehicle. Terrain maps were developed using aerial photos from Google Earth, stitched together using Google SketchUp, and imported into the Gazebo environment. Figure 5.4 shows the three-dimensional rendering of the maps in Google Sketchup.

5.3 Image Registration

Occlusions can be determined by the comparing the rendered camera view and the actual camera input. These images may contain minor discrepancies between the perceived orientation and position of the Gazebo renderings and the camera input. This means that the rendered image may not exactly match camera input and therefore the two images must be registered to each other. Two approaches used for image registration are the Speeded Up Robust Features (SURF) algorithm, as well as the Lucas Kanade algorithm.

5.3.1 Feature Selection

Often times there is too much information within an image, either due to noise or non-unique textures. Therefore, it is beneficial to determine key features, such as corners and edges within the image to use for registration. The corners and

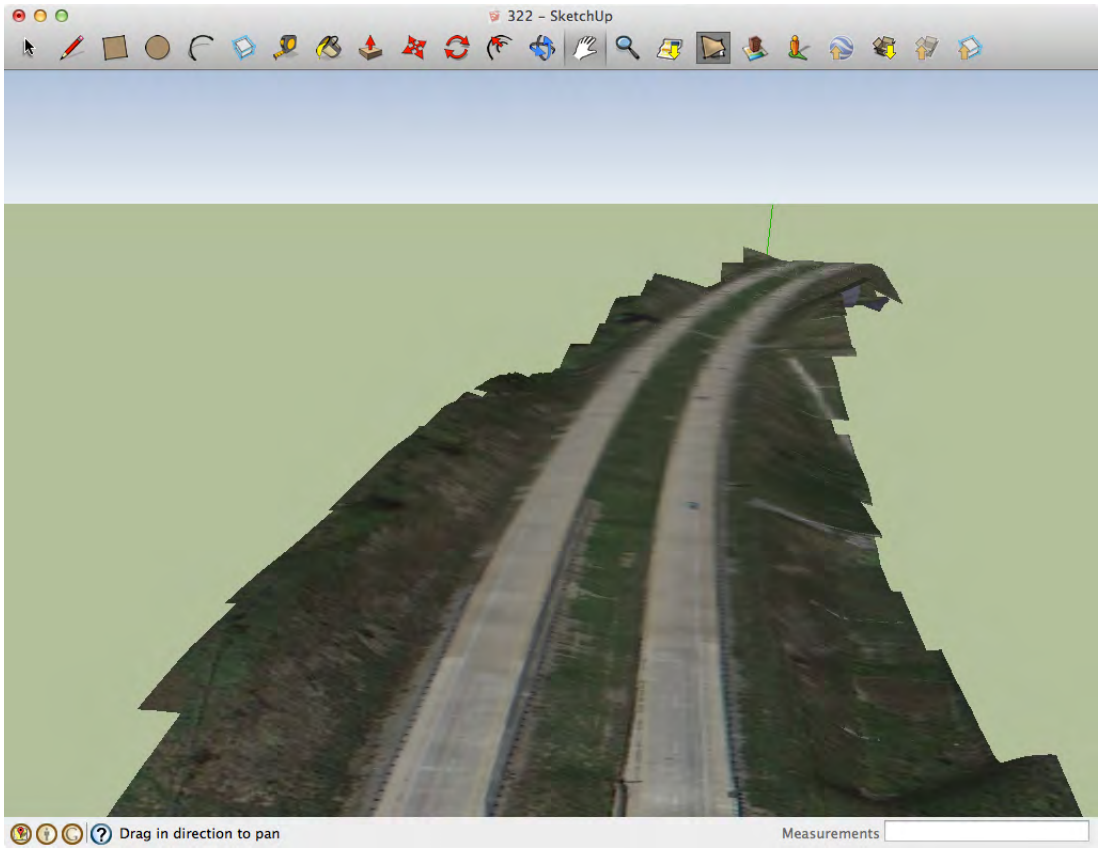

Figure 5.4: 3D map generation in Google Sketchup using Google Earth images.

edges are determined by calculating the eigenvectors and eigenvalues of the covariance matrix, G, of local derivatives within a neighborhood of each pixel. The neighborhood derivatives are calculated through a Sobel operator. [30]

$$G = \begin{bmatrix} \sum I_x^2 & \sum (I_x I_y)^2 \\ \sum (I_x I_y)^2 & \sum I_y^2 \end{bmatrix} \tag{5.6}$$

The minimum eigenvalue of G at each pixel is computed and thresholded based on a maximum allowable eigenvalue. The local maximum of the pixels within the window size are calculated and reduced by non-maximum suppression in the window size neighborhood. [30] Non-maximum suppression searches the adjacent pixels in the direction of the local gradient within the window. Of these selected pixels, the pixel with the highest intensity is determined to be an edge point

and the intensity of the other pixels are set to zero. The algorithm progresses perpendicular to the maximum local gradient of the previous window in search for the next edge pixel. [25] The remaining pixels are considered good features to track. The *goodFeaturesToTrack()* function within the OpenCV library was used to compute the features to be tracked.

5.3.2 SURF & Nearest Neighbor

The Speeded Up Robust Features (SURF) algorithm was used as an initial image registration approach. The algorithm utilizes integral images to determine the image registration. An integral image is defined by the following equations: [31]

$$I(\mathbf{x}) = \sum_{i=0}^{i\leq 0} \sum_{j=0}^{j\leq 0} I(i,j) \tag{5.7}$$

where $\mathbf{x} = (x, y)^T$, a specific pixel location.

Features are determined in the image, such as edges and corners, by summing the 2D Haar wavelet responses of the integral images. Figure 5.5 shows the Haar wavelet filters. The matching of features is determined based on the Euclidean distance between two feature vectors. This algorithm is scale-invariant and can compute the registration translation, rotation, and skew accurately. While the algorithm was successful in registering nearly any two input images, it was computationally slow, running between 0.5 Hz to 0.7 Hz. Figure 5.8 shows the results of the algorithm. [31] The *surf* feature detector functions within the OpenCV library were used to compute the SURF image registration.

5.3.3 Lucas Kanade

The second algorithm used for image registration was the Lucas Kanade method. This algorithm attempts to minimize the image disparities between two images using a similar iterative approach like the Newton-Raphson method. Image $F(x+$

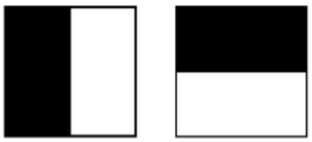

Figure 5.5: 2D Haar wavelet filters used in the SURF algorithm. H. Bay and A. Ess and T.Tuytelaars et al., *Speeded-Up Robust Features (SURF)*. Accessed Apr. 04, 2013.

Figure 5.6: Image to be registered to Figure 5.7.

Figure 5.7: Base image for Figure 5.6 to be registered to. Results can be found in Figure 5.9.

h) represents the image to be registered to, where h represents the image disparity, and image $G(x)$ represents the image to be registered.

The algorithm can be expanded into n-dimensional space, where $\mathbf{x}$ and $\mathbf{h}$ are n-dimensional row vectors. $\mathbf{x}$ represents a vector of corresponding pixel locations in $F(\mathbf{x}+\mathbf{h})$ and $G(\mathbf{x})$. The algorithm attempts to reduce the l_2 norm of the error, $\mathbf{E}$, between the image disparities with respect to $\mathbf{h}$. [32]

$$\mathbf{E} = \sum_x [F(\mathbf{x}+\mathbf{h}) - G(\mathbf{x})]^2 \tag{5.8}$$

By assuming each pixel displacement is within a small neighborhood from image to image, the following first-order linear approximation can be used to represent the image F.

Figure 5.8: Registration of Figure 5.6 and Figure 5.7 using SURF algorithm.

$$F(\mathbf{x}+\mathbf{h}) \approx F(\mathbf{x}) + h\frac{\partial}{\partial \mathbf{x}}F(\mathbf{x}) \tag{5.9}$$

Substituting 5.8 into 5.9, differentiating with respect to $\mathbf{h}$:

$$\begin{aligned} \mathbf{0} &= \frac{\partial}{\partial \mathbf{h}}\mathbf{E} \\ &\approx \frac{\partial}{\partial \mathbf{h}}\sum\nolimits_x [F(\mathbf{x}) + \mathbf{h}\frac{\partial F}{\partial \mathbf{x}} - G(\mathbf{x})]^2 \\ &= \sum\nolimits_x 2\frac{\partial F}{\partial \mathbf{x}}[F(\mathbf{x}) + \mathbf{h}\frac{\partial F}{\partial \mathbf{x}} - G(\mathbf{x})] \end{aligned} \tag{5.10}$$

And solving for the disparity h:

$$\mathbf{h} \approx \Big[\sum\nolimits_x (\frac{\partial F}{\partial \mathbf{x}})^T [G(\mathbf{x}) - F(\mathbf{x})]\Big]\Big[\sum\nolimits_x (\frac{\partial F}{\partial \mathbf{x}})^T (\frac{\partial F}{\partial \mathbf{x}})\Big]^{-1} \tag{5.11}$$

Within Equation 5.11, $G(\mathbf{x}) - F(\mathbf{x})$ represents the error, $(\frac{\partial F}{\partial \mathbf{x}})^T$ represents the point motion, and $[(\frac{\partial F}{\partial \mathbf{x}})^T(\frac{\partial F}{\partial \mathbf{x}})]^{-1}$ represents the local spatial gradient. [30]

The derivation above is only applicable to linear translation between the image disparities. The algorithm can be extended to accommodate any linear transformation including shear, rotation and scaling. Matrix $\mathbf{A}$ from 5.12 represents this generic linear transformation.

$$G(\mathbf{x}) = F(\mathbf{x}\mathbf{A} + \mathbf{h}) \tag{5.12}$$

Thus the l_2 norm equation becomes a function of $\mathbf{A}$:

$$\mathbf{E} = \sum [F(\mathbf{x}\mathbf{A} + \mathbf{h}) - G(\mathbf{x})]^2 \tag{5.13}$$

Utilizing the assumption that pixel motion is small between two sequential images, a linear approximation can be applied to Equation 5.13.

$$F\Big(\mathbf{x}(\mathbf{A} + \Delta\mathbf{A}) + (\mathbf{h} + \Delta\mathbf{h})\Big) \approx F(\mathbf{x}\mathbf{A} + \mathbf{h}) + (\mathbf{x}\Delta\mathbf{A} + \Delta\mathbf{h})\frac{\partial}{\partial \mathbf{x}}F(\mathbf{x}) \tag{5.14}$$

Differentiating Equation 5.14 with respect to $\mathbf{h}$ and $\mathbf{A}$ results in a system of two equations and must be solved simultaneously.

The Lucas Kanade method was successful in registering the images together as seen in Figure 5.9, when the image disparities were sufficiently small and edges were prominent within the two images. The algorithm nearly registered the Gazebo rendering to the camera image, although struggled to find adequate edges on the map rendering image as the lane marker edges were blurred. The computation time for this algorithm was between 16 Hz and 17 Hz, out performing the SURF algorithm significantly. The Lucas Kanade algorithm was implemented in the final occlusion detection algorithm.

Figure 5.9: Registration of Figure 5.6 and Figure 5.7 using Lucas Kanade method.

Figure 5.10: Registration of Figure 5.1 and Figure 5.2 using Lucas Kanade algorithm.

Chapter 6

Occlusion Detection

Following the registration of the map and camera images, occlusions can be determined by the difference between the two images. Where most occlusion detection algorithms require background approximation, the approach of this thesis is to assume that the map representation provides the underlying road and features, less the occlusions. Figure 6.1 presents the image subtraction.

Two image filtering techniques were tested: morphology and basic thresholding. The techniques were evaluated based on the accuracy of occlusion detection and computation time.

6.1 Morphology

A basic morphological algorithm for image filtering is outline in Figure 6.2. First, the image is eroded and thresholded to reduce any spurious artifacts from the image subtraction. The remaining features are dilated to return the features to their original size. This approach was able to detect occlusions at a rate of 7-8 Hz. The result of this can be found in Figure 6.6.

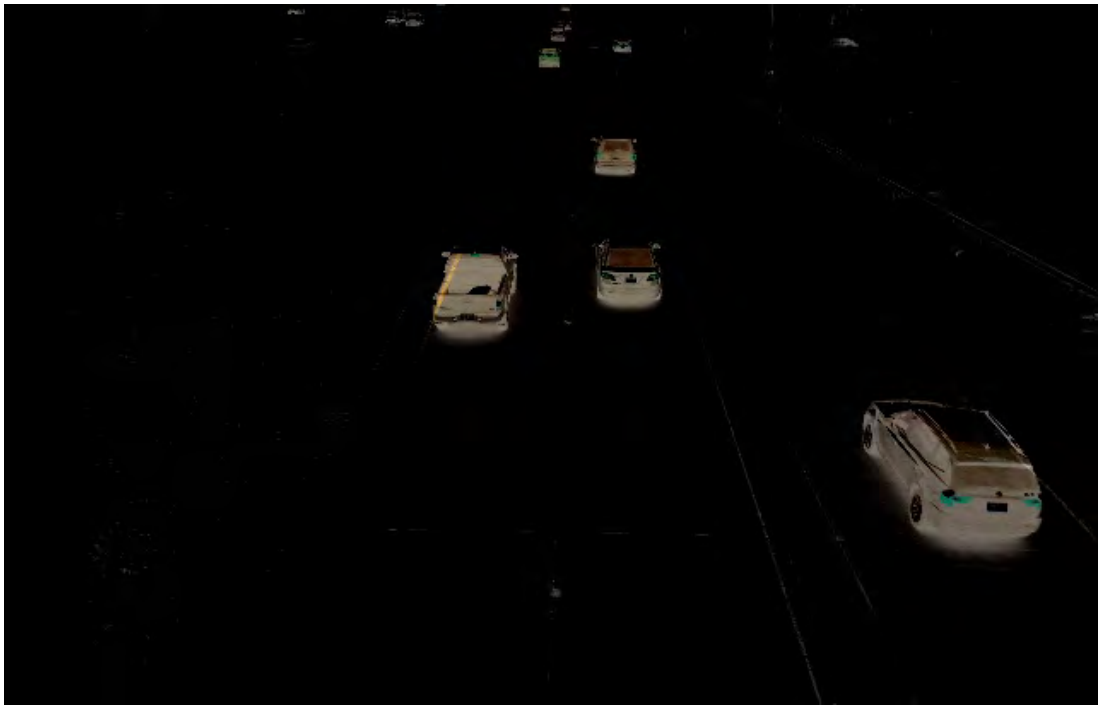

Figure 6.1: Image subtraction after the image registration.

Figure 6.2: Morphological algorithm used for occlusion detection.

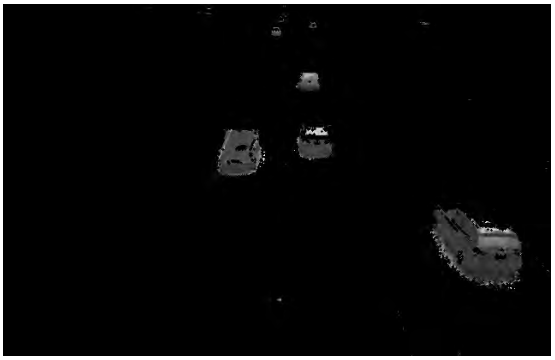

Figure 6.3: Eroded image.

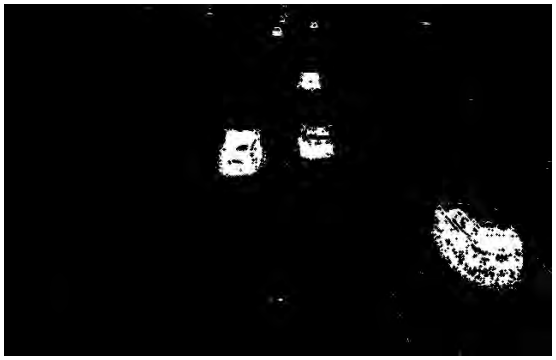

Figure 6.4: Thresholded image.

Figure 6.5: Dilated image.

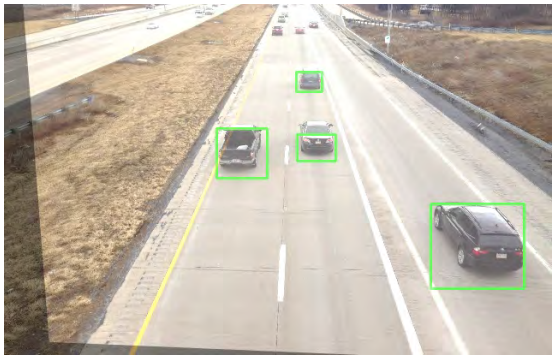

Figure 6.6: Remaining contours thresholded by width, height, and area.

6.2 Basic Thresholding

The second occlusion detection algorithm tested used a basic Gaussian blur and threshold. This approach reduced the computation time slightly, increasing the rate to 11-12 Hz and was successful in determining the occlusions. Figure 6.7 represents the algorithm progression. It should be noted that in Figure 6.10, nearly all the remaining features represent vehicles, but the results are thresholded based on area, height, and width to remove any potential false positives. The result of this approach can be found in Figure 6.11. The algorithm was also run on the Gazebo map rendering images (Figures 5.1 & 5.2) and the results are in Figure 5.10. While the two images did not perfectly register, the occlusion was still able to be detected.

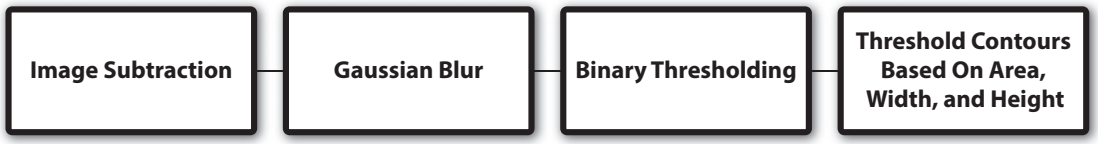

Figure 6.7: Second algorithm used for occlusion detection.

Figure 6.8: Gaussian blurred image.

Figure 6.9: Thresholded image.

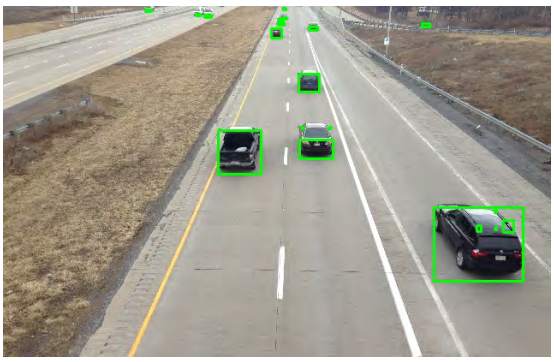

Figure 6.10: All contours.

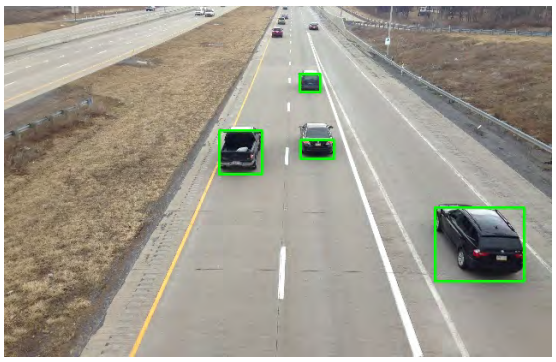

Figure 6.11: Contours thresholded by width, height, and area.

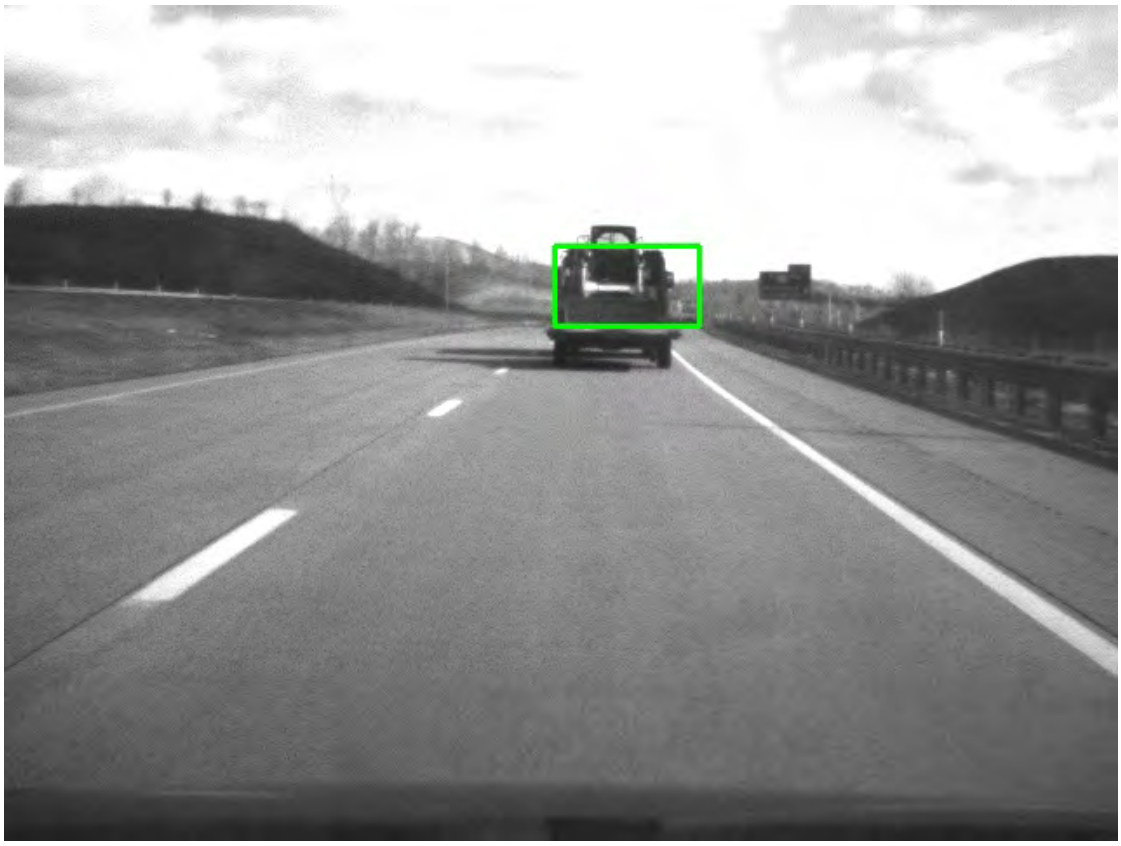

Figure 6.12: Occlusion detection based on Figure 5.10.

Chapter 7

Field Implementation

Often times, most publications on image processing fail to present examples of where their research fails. This is unfortunate, because these failures often present key limitations of the research and sometimes reveal even more important information than the successful attempts! In this chapter, examples of both successes and failures of the algorithm developed are presented and discussed. In each example, the left image is registered to the middle image. The right image is the result of the image registration and occlusion detection algorithms.

Figure 7.1: A successful attempt at registering the left image with the middle image. The right image displays the occlusions found within the image.

Figure 7.1 represents a near perfect registration and occlusion detection. Both images contain defined image features, such as lane markers and road edges. Since there is only a small disparity between the left and middle images, the image registration process only required minor adjustments. The successful image registration

allowed for proper occlusion detection.

Figure 7.2: A failed attempt at registering the left image with the middle image. The right image displays the mis-registration and the determination of some occlusions.

In Figure 7.2, the Lucas Kanade algorithm failed to accurately match the first two images together. This is the result of features in both images not correctly matching. The algorithm registered the left edge of the road near the bottom of the image. Progressing up the image, the left road edge and yellow line deviates from the edge. As seen in the third image, the occlusion detection algorithm still picked up most of the occlusions, but caution should be taken with trusting these results. Occlusions could be missed or even falsely produced by the failure in registering the images.

Figure 7.3: A successful attempt at registering the left image (generated map image) with the middle image (forward-facing camera). The right image displays the occlusion found within the image.

In Figure 7.3, the left image is a rendered map image from Gazebo and is registered to the forward-facing camera image in the middle. The right image shows the successful image registration and occlusion detection. The image registration

relied on the road edges primarily for registration, as the center dashed lane marker does not properly align.

Figure 7.4: A failed attempt at registering the left image with the middle image. The Lucas Kanade algorithm completely failed to find enough feature matches between the left and middle images to produce a registration.

Figure 7.4 displays an attempt to register a generated map image to a forward-facing camera image. The Lucas Kanade algorithm failed to find enough feature matches to produce a homography projection to overlay the images. This shows that the failure to determine any image registration, subsequently causes the occlusion detection process to fail, even though there are occlusions present. As with Figure 7.2, this could be potentially dangerous.

Of the approximately 20 image sequences analyzed, 17 produced unsatisfactory results in both image registration and occlusion detection. In most cases, the image registration failed due to an insufficient amount of matched features between the map and camera images. On real-time video, the algorithm was unable to consistently register the map image. As stated before, this is primarily due to the lack of distinct feature matching, but the algorithms tested in this thesis attempted to register each new image sequence, with no knowledge of the previous registration. By starting with an initial registration guess, only minor adjustments would need to be made.

Chapter 8

Conclusions

The goal of this thesis was to detect a vehicle's position within a road to determine occlusions, based on the matching of map features and features within a forward-facing camera. The results of this thesis are discussed below based on the thesis flow-chart introduced in the Introduction, Chapter 1.

8.1 Vehicle Localization

The data collected from the GPS and MEMS IMU provided sufficient accuracy to localize the vehicle within both the world and the map rendering. Issues arose when the rendered map inaccurately reported the elevation of the mapped terrain, causing a bias in the yaw angle and lateral position of the virtual vehicle with respect to the map. These biases were offset by hard coding values that realigned the system. These biases could be improved with a more accurate representation of the mapped terrain, more specifically the elevation measurements.

8.2 Rendering Of Map Features

Gazebo and ROS provided an quick development process for rendering the maps. This programming environment provided low-level access to the computer hardware, so map rendering was efficient. The accuracy of map features were dependent upon the image quality of the Google Earth aerial photos. Lane features, at times, lacked strong edges which later caused issues with image registration (discussed later).

8.3 Image Registration

The Lucas Kanade image registration accuracy was highly dependent on the quality of the maps generated from Google Earth. If the forward-facing map representation lacked a high contrast between road features, like lane markers and road edges, the registration failed. This meant that the image thresholding values needed to be modified to compensate for the lack of contrast. Also, the accuracy of the Lucas Kanade method depended primarily on small disparities between the real image and the generated image. This meant that algorithm struggled to register the images during real-time video implementations, due to differences in localization of the vehicle in the world and the map. A hybrid approach of using the SURF and Lucas Kanade image registration methods may allow for better image registration: initialize the registration with the SURF method, then make minor updates with the Lucas Kanade method. This would reduce the search space for image registration algorithm, and speed up the registration process so the occlusion detection could be done faster.

8.4 Occlusion Detection

The algorithm described in this thesis was successful in determining occlusions within the forward-facing camera on the vehicle, as well as stationary photos. To more accurately determine occlusions, further modeling of occlusions within a three-dimensional setting, as described in past research, would improve the accuracy of this algorithm. This would allow for the determination of the space they encompass in the three-dimensional world, not just the two-dimensional image plane as developed in this thesis. With the lack of real-time image registration, it was difficult to determine occlusions in real time.

Overall, the map-based approach for occlusion detection provided enticing results. Future research to expand on the map-based research would be to improve algorithms speed and accuracy of both image registration and occlusion detection. The use of a Kalman filter to track occlusions would reduced the search region within the images, as specific regions are occupied by known occlusions. Also, as suggested in the Occlusion Detection section above, the occlusion detection process could be improved by developing a three-dimensional model of the occlusions with respect to the vehicles frame of reference. This research can be extended into the development of autonomous control, as the matching of road features to map features creates a coupling for path planning, collision avoidance, and the ability to see through occlusions for navigational cues displayed on a heads-up display.

Appendix A

Python/OpenCV Code

A.1 Image Acquisition

This code connects to a camera in OpenCV. This code was used to connect to the webcam used for testing.

Listing A.1: Image Acquisition Class

```
#!/usr/bin/python

import cv
import cv2
import numpy as np # for array math

class Camera(object):

  def connectCamera(self,port=0):

    # Connect to the camera
    self.capture = cv2.VideoCapture(port)

  def getFrame(self):

    cam = self.capture

    # Get the next image frame
    ret, img = cam.read()

    return img

if __name__ == '__main__':
  # Create class instances
  cam = Camera()

  # Connect to the camera
  cam.connectCamera(0)

  while True:
    # Grab the current image
    img = cam.getFrame()

    # Show the image
    cv2.imshow('Camera_Output', img)
```

A.2 Camera Calibration

This code calibrates a camera using a checkerboard pattern. It determines the intrinsic and homography matrices, as well as the distortion coefficients.

Listing A.2: Camera Calibration Class

```
#!/usr/bin/python

import cv
import cv2
import numpy as np
import time
from camera import Camera

class Calibration(object):

  # Parameters:
  # - num_of_imgs : How many images of the chessboard do we want to use to calibrate the
      camera? More the better.
  # - square_size : What is the length of one square? Default is 1 in. but divide this by 12 to
      get results in feet. (e.g. 1/12 = 0.0833...)
  # - pattern_size : What are the dimensions of the chessboard? Make sure to only include
      INNER squares, not the outer boundary of squares.
  def calibrate(self,num_of_imgs,square_size=0.0833,pattern_size=[6,9]):

    # Check to see if we already have camera calibration data
    self.findCameraCalibrationFiles()

    # If the user wants to overwrite prior data, we need to calibrate again so this validates
    if (self.ovrt_int == 1 or self.ovrt_dst == 1):

      # Create camera class instance
      cam = Camera()

      # Connect to camera
      cam.connectCamera()

      # We're ready to calibrate
      print "Please place the chessboard in front of the camera."

      # Set some initial parameters
      self.pattern_width = pattern_size[0] # number of chessboard squares in one direction (e.g
          . 6)
      self.pattern_height = pattern_size[1] # number of chessboard squares in the other
          direction (e.g. 9)
      self.pattern_size = (self.pattern_width,self.pattern_height) # create an array of the
          pattern size (e.g. (6,9))
      self.pattern_n = np.prod(self.pattern_size) # total number of corners to detect (e.g. 6*9
          = 54)

      # Allocate memory for camera parameters
      if(self.ovrt_int == 1):
        self.intrinsic_matrix = np.zeros((3, 3))
      if(self.ovrt_dst == 1):
        self.distortion_coeffs = np.zeros((5,1))

      # Allocate memory for camera matrices
      self.pattern_pts = np.zeros((self.pattern_n,3),np.float32)
      self.pattern_pts[:,:2] = np.indices(pattern_size).T.reshape(-1,2)
      self.pattern_pts *= square_size # Uses real world distances

      # This while loop:
      # - Grabs a new image
      # - Finds the chessboard corners
      # - If we found enough corners, add it to the corner array
```

```
        # - Increase successes by 1, increase until we reached num_of_imgs
        successes = 0
        self.object_pts = []
        self.image_pts = []
        while (successes < num_of_imgs):

            # Grab the current image
            #img = cam.getFrame()
            img = cv2.imread('distorted-img.jpg') # DELETE

            # Get the size of the image
            size = img.shape[1],img.shape[0]

            # Show the image
            cv2.imshow('Distorted', img)

            # Get the corners of the chessboard
            corners = self.findChessboard(img)

            # Do we have any corners?
            if(corners is not None):

                # If so, do we have enough?
                if(len(corners) == self.pattern_n):

                    # Add the image points to the total image array
                    self.image_pts.append(corners.reshape(-1,2))
                    # Add the object points to the total object array
                    self.object_pts.append(self.pattern_pts)
                    successes += 1
                    print "Chessboards found:", successes
                    time.sleep(2)

        if(successes == num_of_imgs):

            # We got enough images to get the camera parameters
            print "Obtaining calibration parameters..."

            # This is the calibrateCamera function.
            rms, intrinsic_matrix, distortion_coeffs, rvecs, tvecs = cv2.calibrateCamera(
                self.object_pts,
                self.image_pts,
                size
            )

            # Save the camera parameters
            if(self.ovrt_dst == 1):
                self.distortion_coeffs = distortion_coeffs
                cv2.cv.Save("distortion.xml",cv.fromarray(self.distortion_coeffs))
            if(self.ovrt_int == 1):
                self.intrinsic_matrix = intrinsic_matrix
                cv2.cv.Save("intrinsics.xml",cv.fromarray(self.intrinsic_matrix))

            # Print out the camera data
            print "RMS:", rms
            print "Intrinsic Matrix:\n", self.intrinsic_matrix
            print "Distortion Coefficients: ", self.distortion_coeffs.ravel()

            cv.DestroyWindow('Distorted')

    def findChessboard(self,img):

        # Convert image to grayscale
        gray_img = cv2.cvtColor(img,cv.CV_BGR2GRAY)

        # Find the chessboard corners with the findChessboardCorners function
        # Function requirements:
        # image : image to find the chessboard
```

```
    # patternSize : chessboard width x chessboard height e.g. 5x7
    # flags (optional) : adaptive thresholding
    #   quads filter
    #   fast check
    self.found, corners = cv2.findChessboardCorners(
                            image = gray_img,
                            patternSize = self.pattern_size,
                            flags = cv2.CALIB_CB_ADAPTIVE_THRESH|cv2.
                                CALIB_CB_FILTER_QUADS|cv2.CALIB_CB_FAST_CHECK
                        )
        # Did we find any corners? If not, return None
    if not self.found:
      return None
        # If we did find enough corners (board_size), let's be more accurate (subpixel accuracy)
        # Function requirements :
        # image : original image from before
        # corners : the corners we found above from findChessboardCorners
        # winSize :
        # zeroZone :
        # criteria :
    cv2.cornerSubPix(
            image = gray_img,
            corners = corners,
            winSize = (11,11),
            zeroZone = (-1,-1),
            criteria = (cv.CV_TERMCRIT_EPS|cv.CV_TERMCRIT_ITER,30,0.1)
    )
    # We were successful at finding corners, now lets return them
    return corners
def undistort(self,img,intrinsic_matrix=None,distortion_coeffs=None):
    # Undistort the image
    if(intrinsic_matrix == None and distortion_coeffs == None):
      img = cv2.undistort(img,self.intrinsic_matrix,self.distortion_coeffs)
    else:
      img = cv2.undistort(img,intrinsic_matrix,distortion_coeffs)
    return img
def findHomography(self,square_size=0.0833,pattern_size=[6,9]):
    # See if we already have a homography matrix ('homography.xml') in the folder.
    self.findHomographyFiles()
    # See if we already have an intrinsic matrix ('intrinsic.xml'), and
    # the distortion coefficient matrix ('distortion.xml') in the folder.
    self.findCameraCalibrationFiles()
    if(self.ovrt_hom == 1):
      # Create camera class instance
      cam = Camera()
      # Connect to camera
      cam.connectCamera()
      # We're ready to calibrate
      print "Please place the chessboard in front of the camera."
      # Set some initial parameters
      self.pattern_width = pattern_size[0] # number of chessboard squares in one direction (e.g
          . 6)
```

```
self.pattern_height = pattern_size[1] # number of chessboard squares in the other
      direction (e.g. 9)
self.pattern_size = (self.pattern_width,self.pattern_height) # create an array of the
      pattern size (e.g. (6,9))
self.pattern_n = np.prod(self.pattern_size) # total number of corners to detect (e.g. 6*9
      = 54)

# Allocate memory for camera parameters
if(self.ovrt_hom == 1):
  self.homography_matrix = np.zeros((3, 3))

# Allocate memory for camera matrices
self.pattern_pts = np.zeros((self.pattern_n,2),np.float32)
self.pattern_pts[:,:2] = np.indices(pattern_size).T.reshape(-1,2)
self.pattern_pts *= square_size # Uses real world distances

# This while loop:
# - Grabs a new image
# - Finds the chessboard corners
# - If we found enough corners, add it to the corner array
# - Increase successes by 1, increase until we reached num_of_imgs
successes = 0
self.object_pts = []
self.image_pts = []
while (successes < 1):

  # Grab the current image
  img = cam.getFrame()

  # Get the size of the image
  size = img.shape[1],img.shape[0]

  # Undistort the image
  img = self.undistort(img)

  # Show the image
  cv2.imshow('Undistorted', img)

  # Get the corners of the chessboard
  corners = self.findChessboard(img)

  # Do we have any corners?
  if(corners is not None):

    # If so, do we have enough?
    if(len(corners) == self.pattern_n):

      # Add the image points to the total image array
      self.image_pts.append(corners.reshape(-1,2))
      # Add the object points to the total object array
      self.object_pts.append(self.pattern_pts)
      successes += 1
      print "Chessboards_found:", successes
      time.sleep(2)

if(successes == 1):

  # We got enough images to get the homography matrix
  print "Obtaining_homography_matrix..."

  print self.object_pts[0]
  print self.image_pts[0]
  # This is the findHomography function.
  homography_matrix, mask = cv2.findHomography(
    self.object_pts[0],
    self.image_pts[0],
```

```
            cv2.RANSAC
        )
        # Save the homography matrix
        if(self.ovrt_hom == 1):
            self.homography_matrix = homography_matrix
            cv2.cv.Save("homography.xml",cv.fromarray(self.homography_matrix))
        # Print out the homography matrix
        print "Homography_Matrix:\n", self.homography_matrix

        cv.DestroyWindow('Undistorted')
def homographyProjection(self,img,homography_matrix,inverse=False):
    # Projects the input image with the supplied homography matrix.
    # Get the size of the input image
    size = img.shape[1],img.shape[0]
    # This corrects the output position on the screen. This needs to be fixed. TODO
    # DPI correction? Aspect ratio.
    T = np.mat([[1, 0, -500],[0, 1, -180],[0, 0, 200]])
    T_inv = np.linalg.inv(T)
    # Warp the image
    if(inverse == False):
        img = cv2.warpPerspective(img,np.dot(homography_matrix,T),size,flags=cv2.
            WARP_INVERSE_MAP | cv2.INTER_LINEAR)
    else:
        img = cv2.warpPerspective(img,np.dot(T_inv,homography_matrix),size,flags=cv2.
            WARP_INVERSE_MAP | cv2.INTER_LINEAR)
    return img

def intrinsic(self):
    # Load the intrinsic matrix if it exists.
    while True:
        try:
            self.intrinsic_matrix = cv2.cv.Load('intrinsics.xml')
            self.intrinsic_matrix = np.asarray(self.intrinsic_matrix[:,:])
            print "Intrinsic_Matrix:\n", self.intrinsic_matrix
            break
        except TypeError:
            break
    return self.intrinsic_matrix
def distortion(self):
    # Load the distortion coefficients if it exists.
    while True:
        try:
            self.distortion_coeffs = cv2.cv.Load('distortion.xml')
            self.distortion_coeffs = np.asarray(self.distortion_coeffs[:,:])
            print "Distortion_Coefficients:_", self.distortion_coeffs.ravel()
            break
        except TypeError:
            break
    return self.distortion_coeffs
def homography(self):
    # Load the homography matrix if it exists.
    while True:
        try:
```

```
        self.homography_matrix = cv2.cv.Load('homography.xml')
        self.homography_matrix = np.asarray(self.homography_matrix[:,:])
        print "Homography_Matrix:", self.homography_matrix
        break
      except TypeError:
        break
    return self.homography_matrix

  def invHomography(self):

    # Load the homography matrix if it exists, then invert it.

    while True:
      try:
        self.inv_homography = cv2.cv.Load('homography.xml')
        self.inv_homography = np.asarray(self.inv_homography[:,:])
        self.inv_homography = np.linalg.inv(self.inv_homography)
        print "Inverse_Homography_Matrix:", self.inv_homography
        break
      except TypeError:
        break
    return self.inv_homography

  def findCameraCalibrationFiles(self):

    ### Intrinsic Matrix ###

    # Check to see if we have a 'intrinsics.xml' file already in the folder
    # Otherwise, we need to create one.

    while True:
      try:
        self.intrinsic_matrix = cv2.cv.Load('intrinsics.xml')
        self.intrinsic_matrix = np.asarray(self.intrinsic_matrix[:,:])
        print "Intrinsic_Matrix:\n", self.intrinsic_matrix
        self.ovrt_int = raw_input('Would_you_like_to_overwrite_the_intrinsic_matrix?_y/n_')
        while True:
          if(self.ovrt_int == 'y' or self.ovrt_int == 'Y'):
            self.ovrt_int = 1
            break
          elif(self.ovrt_int == 'n' or self.ovrt_int == 'N'):
            self.ovrt_int = 0
            break
          self.ovrt_int = input('Would_you_like_to_overwrite_the_intrinsic_matrix?_y/n_')
        break
      except TypeError:
        self.intrinsic_matrix = None
        break

    ### Distortion Coefficients ###

    # Check to see if we have a 'distortion.xml' file already in the folder
    # Otherwise, we need to create one.
    while True:
      try:
        self.distortion_coeffs = cv2.cv.Load('distortion.xml')
        self.distortion_coeffs = np.asarray(self.distortion_coeffs[:,:])
        # self.distortion_coeffs[0,0] = 1
        # self.distortion_coeffs[0,1] = -1
        # self.distortion_coeffs[0,2] = 0
        # self.distortion_coeffs[0,3] = 0
        # self.distortion_coeffs[0,4] = 0
        print "Distortion_Coefficients:", self.distortion_coeffs.ravel()
        self.ovrt_dst = raw_input('Would_you_like_to_overwrite_the_distortion_coefficients?_y/n
            _')
```

```
            while True:
              if(self.ovrt_dst == 'y' or self.ovrt_dst == 'Y'):
                self.ovrt_dst = 1
                break
              elif(self.ovrt_dst == 'n' or self.ovrt_dst == 'N'):
                self.ovrt_dst = 0
                break
              self.ovrt_dst = input('Would you like to overwrite the distortion coefficients? y/n '
                  )
              break
            except TypeError:
              self.distortion_coeffs = None
              break
    def findHomographyFiles(self):
        # Check to see if we have a 'homography.xml' file already in the folder
        # Otherwise, we need to create one.
        while True:
          try:
            self.homography_matrix = cv2.cv.Load('homography.xml')
            self.homography_matrix = np.asarray(self.homography_matrix[:,:])
            print "Homography Matrix: ", self.homography_matrix
            self.ovrt_hom = raw_input('Would you like to overwrite the homography matrix? y/n
                ')
            while True:
              if(self.ovrt_hom == 'y' or self.ovrt_hom == 'Y'):
                self.ovrt_hom = 1
                break
              elif(self.ovrt_hom == 'n' or self.ovrt_hom == 'N'):
                self.ovrt_hom = 0
                break
              self.ovrt_hom = input('Would you like to overwrite the homography matrix? y/n ')
              break
            except TypeError:
              self.homography_matrix = None
              break
if __name__ == '__main__':
    # Create class instances
    cal = Calibration()
    cam = Camera()
    # Connect to the camera
    cam.connectCamera()
    # Calibrate the camera
    cal.calibrate(num_of_imgs=10,square_size=0.08333,pattern_size=[6,9])
    # For homography matrix calculation, comment the above command and uncomment below
    cal.findHomography(square_size=0.08333,pattern_size=[6,9])
    while True:
        # Grab the current image
        img = cam.getFrame()
        # Undistort the image
        original = cal.undistort(img)
        # Use homography for inverse perspective
        original = cal.homographyProjection(img,self.homography_matrix)
        # Show the image
        cv2.imshow('Undistorted Image', original)
```

A.3 Lucas Kanade Image Registration

This code using the Lucas Kanade method for image registration. The output is one image of the two input images registered and overlaid.

Listing A.3: Lucas Kanade Image Registration Class

```
#!/usr/bin/python

import cv2
import numpy as np

# Lucas Kanade Parameters
# - winSize : Window size for local variations
# - maxLevel : Number of pyramidal steps for the algorithm
# - criteria :
lk_params = dict( winSize = (5, 5),
                  maxLevel = 5, # 1
                  criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT,
                      10, 0.03))

# Good Features To Track Parameters
# - maxCorners : Total number of corners to return
# - quality level :
# - minDistance : Minimum distance between two good features
# - blockSize : Window size for iterating across the image
feature_params = dict( maxCorners = 1000, # 2000
                       qualityLevel = 0.01,
                       minDistance = 8, # 8
                       blockSize = 19 )

# Image Registration Class
class ImageRegistration:
  def __init__(self):

    # Homography/Ransac Settings
    self.use_ransac = True
    self.ransac_iter = 10.0

    # Canny Settings for camera
    self.cam_canny_min_threshold = 50
    self.cam_canny_max_threshold = 150

    # Canny Settings for map
    self.map_canny_min_threshold = 50 # 250
    self.map_canny_max_threshold = 150 # 350

    # Gaussian Blur Settings for camera
    self.cam_gaussian_blur_size = 3
    self.cam_gaussian_blur_sigma = 1

    # Gaussian Blur Settings for map
    self.map_gaussian_blur_size = 3
    self.map_gaussian_blur_sigma = 1

    # Lucas Kanade Settings
    self.back_threshold = 1.0

    # Overlay Settings
    self.overlay_color = (0,0,255)

  def checkedTrace(self, cam_img, map_img, p0, back_threshold = 1.0):
    # cam_img : Camera Image
    # map_img : Map Image
    # p0 : Good feature points found in the camera image
    # back_threshold : Minimum distance between forward and backward flow vectors

    # Calculate forward and backward Lucas Kanade Optical Flow
```

```
        p1, st, err = cv2.calcOpticalFlowPyrLK(cam_img, map_img, p0, None, **lk_params)
        p0r, st, err = cv2.calcOpticalFlowPyrLK(map_img, cam_img, p1, None, **lk_params)

        # Calculate the max distance between the original and final vectors
        d = abs(p0-p0r).reshape(-1, 2).max(-1)

        # Determine "good" point correspondents by thresholding (back_threshold)
        status = d < back_threshold

        # Return the new points and whether they were good or bad.
        return p1, status

    def filterCamImage(self,img):

        # Gray scale the image
        img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        # Apply a Gaussian blur
        img_gaus = cv2.GaussianBlur(img_gray, (self.cam_gaussian_blur_size, self.
            cam_gaussian_blur_size), sigmaX=self.cam_gaussian_blur_sigma, sigmaY=self.
            cam_gaussian_blur_sigma)

        # Apply a Canny filter for edge detection
        img_canny = cv2.Canny(img_gaus, threshold1=self.cam_canny_min_threshold, threshold2=
            self.cam_canny_max_threshold)

        return img_canny

    def filterMapImage(self,img):

        # Gray scale the image
        img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        # Apply a Gaussian blur
        img_gaus = cv2.GaussianBlur(img_gray, (self.map_gaussian_blur_size, self.
            map_gaussian_blur_size), sigmaX=self.map_gaussian_blur_sigma, sigmaY=self.
            map_gaussian_blur_sigma)

        # retval, img_thresh = cv2.threshold(img_gray, 245, 255, cv2.THRESH_BINARY)

        # Apply a Canny filter for edge detection
        img_canny = cv2.Canny(img_gaus, threshold1=self.map_canny_min_threshold, threshold2=
            self.map_canny_max_threshold)

        return img_canny

    def registration(self, map_img, cam_img):

        # Filter both images (Gaussian/Canny)
        map_filt = self.filterMapImage(map_img)
        cam_filt = self.filterCamImage(cam_img)

        # Find good feature points in the camera image
        p0 = cv2.goodFeaturesToTrack(cam_filt, **feature_params)

        p1 = p0

        # Use the Lucas Kanade algorithm to determine the optical flow the the features
        p2, trace_status = self.checkedTrace(cam_filt, map_filt, p1, self.back_threshold)

        # Store the "good" new features
        p1 = p2[trace_status].copy()

        # Only keep the good "old" features.
        p0 = p0[trace_status].copy()

        print trace_status, len(p0), len(p1)

        # Calculate the homography between the the map features and the camera features
        if len(p0) >= 4:
```

```
        # Calculate the homograpy matrix
        H, status = cv2.findHomography(p1, p0, cv2.RANSAC, self.ransac_iter)
        print H
        # Get the width and height of the map image
        h, w = map_img.shape[:2]
        # Warp the map image to match with the camera image using the homography
        map_reg = cv2.warpPerspective(map_img, H, (w, h))
        # Create tint overlay for registered map
        overlay = np.zeros((h,w,3), np.uint8)
        cv2.rectangle(overlay,(0,0),(0+w,0+h),self.overlay_color,-1)
        overlay = cv2.warpPerspective(overlay, H, (w, h))
    else:
        map_reg = None
        overlay = None
        status = None
    return map_reg, overlay, status
```

A.4 SURF Image Registration

This code uses the Speeded Up Robust Features algorithm for image registrations. The output is one image of the two input images registered and overlaid.

Listing A.4: SURF Image Registration Class

```
import cv2
import numpy as np
import time
import sys
from camera import Camera

class SurfRegistration(object):

    def registration(self,needle,haystack):

        # Input images:
        # haystack : image to be searched
        # needle : image to find
        self.needle = needle
        self.haystack = haystack
        # Grayscale the images
        ngrey = cv2.cvtColor(self.needle, cv2.COLOR_BGR2GRAY)
        hgrey = cv2.cvtColor(self.haystack, cv2.COLOR_BGR2GRAY)
        # build feature detector and descriptor extractor
        detector = cv2.FeatureDetector_create("SURF")
        descriptorExtractor = cv2.DescriptorExtractor_create("SURF")
        hkeypoints = detector.detect(hgrey)
        nkeypoints = detector.detect(ngrey)
        start = default_timer()
        (hkeypoints, hdescriptors) = descriptorExtractor.compute(hgrey,hkeypoints)
        (nkeypoints, ndescriptors) = descriptorExtractor.compute(ngrey,nkeypoints)
        # extract vectors of size 64 from raw descriptors np arrays
        rowsize = 128
        hrows = np.array(hdescriptors, dtype = np.float32).reshape((-1, rowsize))
        nrows = np.array(ndescriptors, dtype = np.float32).reshape((-1, rowsize))
        # kNN training - learn mapping from hrow to hkeypoints index
```

```
        samples = hrows
        responses = np.arange(len(hrows), dtype = np.float32)
        knn = cv2.KNearest()
        knn.train(samples,responses,maxK=10)

        good = 0
        bad = 0
        hpoints = []
        npoints = []
        # retrieve index and value through enumeration
        for i, descriptor in enumerate(nrows):
            # Reorganize the descriptor array
            descriptor = np.array(descriptor, dtype = np.float32).reshape((1, 128))

            # Find the nearest neighbors
            retval, results, neigh_resp, dists = knn.find_nearest(descriptor, 1)

            # Separate out the results and the distances between points.
            res, dist = int(results[0][0]), dists[0][0]

            # If the distances is less than this value, we like it...
            if dist < 0.1:
                # draw matched keypoints in red color
                color = (0, 0, 255)
                good += 1
            # Else we hate it
            else:
                # draw unmatched in blue color
                color = (255, 0, 0)
                bad += 1
            # draw matched key points on haystack image
            x,y = hkeypoints[res].pt
            center = (int(x),int(y))
            # cv2.circle(self.haystack,center,2,color,-1)
            hpoints = np.append(hpoints,(x,y))
            # draw matched key points on needle image
            x,y = nkeypoints[i].pt
            center = (int(x),int(y))
            # cv2.circle(self.needle,center,2,color,-1)
            npoints = np.append(npoints,(x,y))

        # Reshape the good corresponding point arrays for the homography function below.
        hpoints = hpoints.reshape(-1,2)
        npoints = npoints.reshape(-1,2)

        # Calculate the homograpy matrix
        H, status = cv2.findHomography(npoints, hpoints, cv2.RANSAC, 10.0)

        # Get the width and height of the map image
        h, w = self.needle.shape[:2]

        # Warp the map image to match with the camera image using the homography
        self.needle = cv2.warpPerspective(self.needle, H, (w, h))

        # Create tint overlay for registered map
        overlay = np.zeros((h,w,3), np.uint8)
        cv2.rectangle(overlay,(0,0),(0+w,0+h),(0,0,255),-1)
        overlay = cv2.warpPerspective(overlay, H, (w, h))

        # How good did we do?
        accuracy = round(100*good/float(good+bad),2)

        return self.needle, self.haystack, accuracy, overlay
if __name__ == '__main__':
```

```
    # Declare the input images.
    haystack =cv2.imread('haystack.jpg')
    needle = cv2.imread('needle.jpg')

    # Initialize the SurfRegistration class
    reg = SurfRegistration()

    while True:
        # Register the images
        needle, haystack, accuracy, overlay = reg.registration(needle, haystack)

        # Overlay the results
        cam_und = cv2.addWeighted(haystack, 0.7, needle, 0.7, 0.0)
        cam_und = cv2.addWeighted(overlay, 0.2, cam_und, 1 - 0.2, 0.0)

        # Show the results
        cv2.imshow('Haystack',cam_und)
```

A.5 Occlusion Detection

This code determines occlusions within an image, based on the input of two registered images.

Listing A.5: Occlusion Detection Class

```
#!/usr/bin/python

import cv2
import numpy as np
from projectPoint import ProjectPoint

class OcclusionDetection:
  def __init__(self):

    self.cvt = ProjectPoint()

    # Car Modeling Parameters
    self.car_box_color = (0,255,0)
    self.car_width_min = 12
    self.car_width_max = 120
    self.car_height_min = 12
    self.car_height_max = 120
    self.car_area_min = 30
    self.car_area_max = 20000
    # Truck Modeling Parameters
    self.truck_box_color = (255,0,0)
    self.truck_width_min = 50
    self.truck_width_max = 200
    self.truck_height_min = 30
    self.truck_height_max = 200
    self.truck_area_min = 30
    self.truck_area_max = 200
    # Pedestrian Modeling Parameters
    self.pedestrian_box_color = (0,0,255)
    self.pedestrian_width_min = 30
    self.pedestrian_width_max = 200
    self.pedestrian_height_min = 30
    self.pedestrian_height_max = 200
    self.pedestrian_area_min = 100
    self.pedestrian_area_max = 300
    ## Thresholding Settings ##
    # Regular Thresholding
```

```
    self.threshold_min = 80 # 50
    self.threshold_max = 255 # 255

    # Gaussian Blur Settings
    self.gaussian_blur_size = 3 # 7
    self.gaussian_blur_sigma = 1 # 3

  def filterImage(self, img):

    # Grayscale the image
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Morphological Filtering
    # element = cv2.getStructuringElement(cv2.MORPH_CROSS,(3,3))
    # img_erode = cv2.erode(img_gray,element)
    # retval, img_thresh = cv2.threshold(img_erode,self.threshold_min,self.threshold_max,cv2.
        THRESH_BINARY)
    # img_dilate = cv2.dilate(img_thresh,element)

    # Basic Thresholding
    img_gaus = cv2.GaussianBlur(img_gray, (self.gaussian_blur_size,self.gaussian_blur_size),
        sigmaX=self.gaussian_blur_sigma, sigmaY=self.gaussian_blur_sigma)
    retval, img_thresh = cv2.threshold(img_gaus, self.threshold_min, self.threshold_max, cv2.
        THRESH_BINARY)

    return img_thresh

  def subtractImages(self, map_img, cam_img):

    sub = cv2.subtract(map_img,cam_img)
    return sub

  def findContours(self, img):

    contours, hierarchy = cv2.findContours(img,cv2.RETR_EXTERNAL,cv2.
        CHAIN_APPROX_SIMPLE)
    return contours

  def drawOcclusions(self, img, contours):

    for cnt in contours:
      # status = True
      cnt, status = self.groupObjects(cnt)

      if status is True:
        # Properties of the contour
        x,y,w,h = cv2.boundingRect(cnt)
        left = tuple(cnt[cnt[:,:,0].argmin()][0])
        right = tuple(cnt[cnt[:,:,0].argmax()][0])
        top = tuple(cnt[cnt[:,:,1].argmin()][0])
        bottom = tuple(cnt[cnt[:,:,1].argmax()][0])
        x_middle = int((right[0] - left[0])/2 + left[0])
        y_middle = int(bottom[1])

        # Uncomment to show all contours
        # cv2.rectangle(img,(x,y),(x+w,y+h),self.pedestrian_box_color,2)

        # Corners
        top_right = np.array([right[0], top[1]])
        bottom_right = np.array([right[0], bottom[1]])
        bottom_left = np.array([left[0], bottom[1]])
        top_left = np.array([left[0], top[1]])
        center = np.array([x_middle, y_middle])

        # Convert points/measurements to world coordinates
        top_right_world = self.cvt.project2Dto2D(top_right)
```

```
        bottom_right_world = self.cvt.project2Dto2D(bottom_right)
        bottom_left_world = self.cvt.project2Dto2D(bottom_left)
        top_left_world = self.cvt.project2Dto2D(top_left)
        center_world = self.cvt.project2Dto2D(center)
        w_world = bottom_right_world[0] - bottom_left_world[0]
        h_world = bottom_left_world[1] - top_left_world[1]
        a = w_world*h_world # Area

        # TODO Can be removed
        w_world = bottom_right[0] - bottom_left[0]
        h_world = bottom_left[1] - top_left[1]
        a = w_world*h_world # Area

        #print w_world, h_world

        x1 = bottom_left_world[0]
        y1 = bottom_left_world[1]
        w1 = w_world
        h1 = h_world
        # cv2.rectangle(img,(x1,y1),(x1+w1,y1+h1),self.car_box_color,2)

        # Check to see if the contour is a car
        if( w_world > self.car_width_min and w_world < self.car_width_max and
          h_world > self.car_height_min and h_world < self.car_height_max and
          a > self.car_area_min and a < self.car_area_max):

          cv2.rectangle(img,(x,y),(x+w,y+h),self.car_box_color,2)

        # Check to see if the contour is a truck
        if( w_world > self.truck_width_min and w_world < self.truck_width_max and
          h_world > self.truck_height_min and h_world < self.truck_height_max and
          a > self.truck_area_min and a < self.truck_area_max):

          cv2.rectangle(img,(x,y),(x+w,y+h),self.truck_box_color,2)

        # Check to see if the contour is a pedestrian
        if( w_world > self.pedestrian_width_min and w_world < self.pedestrian_width_max
            and
          h_world > self.pedestrian_height_min and h_world < self.pedestrian_height_max and
          a > self.pedestrian_area_min and a < self.pedestrian_area_max):

          cv2.rectangle(img,(x,y),(x+w,y+h),self.pedestrian_box_color,2)

    return img

def groupObjects(self, contour):

    hull = cv2.convexHull(contour)

    status = cv2.isContourConvex(hull)

    return hull, status

def findOcclusions(self, map_img, cam_img):

    sub_img = self.subtractImages(map_img, cam_img)

    sub_filt = self.filterImage(sub_img)

    contours = self.findContours(sub_filt)

    cam_occ = self.drawOcclusions(cam_img, contours)

    return cam_occ
```

Bibliography

[1] Zhang, W. and Q. Wu (2008) "Multilevel framework to detect and handle vehicle occlusion," *. . . , IEEE Transactions on*, **9**(1), pp. 161–174.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4445682

[2] Ghasemi, A. and R. Safabakhsh (2012) "A real-time multiple vehicle classification and tracking system with occlusion handling," *2012 IEEE 8th International Conference on Intelligent Computer Communication and Processing*, pp. 109–115.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6356172

[3] Chen, C. and S. Liu (2012) "Detection and Segmentation of Occluded Vehicles Based on Skeleton Features," *2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control*, pp. 1055–1059.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6429085

[4] Luo, J. and J. Zhu (2010) "Improved video-based vehicle detection methodology," *. . . and Information Technology (ICCSIT), 2010 3rd . . .*, pp. 602–606.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5565052

[5] Pang, C. (2004) "A novel method for resolving vehicle occlusion in a monocular traffic-image sequence," *. . . , IEEE Transactions on*.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1331384

[6] Stauffer, C. and W. Grimson (1999) "Adaptive background mixture models for real-time tracking," *Proceedings. 1999 IEEE Computer Society*

Conference on Computer Vision and Pattern Recognition (Cat. No PR00149), pp. 246–252.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=784637

[7] Gupte, S., O. Masoud, R. Martin, and N. Papanikolopoulos (2002) "Detection and classification of vehicles," *IEEE Transactions on Intelligent Transportation Systems*, **3**(1), pp. 37–47.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=994794

[8] Kesling, P. (1963), "A New Driving Simulator Including An Interactive Intelligent Trafic Environment," .
URL http://www.google.com/patents?hl=en&lr=&vid=USPAT3085336&id=GoRGAAAAEBAJ&oi=fnd&dq=Peter+C.&printsec=abstract

[9] Hahn, S. (1993) "SWITCHING BETWEEN AUTONOMOUS AND CONVENTIONAL CAR DRIVING: A SIMULATOR STUDY," *Intelligent Vehicles Symposium (1993: Tokyo, Japan). ...*, pp. 25–30.
URL http://trid.trb.org/view.aspx?id=640591

[10] Ahn, H. B., J. H. Ku, B. H. Cho, H. Kim, H. J. Jo, and J. M. Lee (2001) "THE DEVELOPMENT OF VIRTUAL REALITY DRIVING 1SIMULATORFOR Scenario -," , pp. 3780–3783.

[11] Evans, R. W., A. P. Ramsbottom, and D. W. Sheel (1989) "Head-up Displays in Motor Cars," *Holographic Systems, Components and Applications*, pp. 56–62.
URL hhttp://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=51792

[12] Fouladinejad, N. (2011) "Modeling virtual driving environment for a driving simulator," *... System, Computing and ...*, (Vdm), pp. 27–32.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6190490

[13] Bertollini, G., C. Johnston, and e. a. Kuiper, J (1994) *The General Motors Driving Simulator*.
URL http://papers.sae.org/940179/

[14] Tonnis, M., C. Lange, and G. Klinker (2007) "Visual Longitudinal and Lateral Driving Assistance in the Head-Up Display of Cars," *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, pp. 1–4.

URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4538831

[15] BMW (2013), "BMW Head-Up Display." .
URL http://www.bmw.com/com/en/insights/technology/technology_guide/articles/head_up_display.html

[16] Adam, E. (1993) "Fighter Cockpits Of The Future," *Digital Avionics Systems Conference, 1993. 12th ...*, pp. 318–323.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=283529

[17] Merryweather, R. A. (1990) "Automotive head-up display," *US Patent ...*, (1), pp. 1–4.
URL http://www.google.com/patents?hl=en&lr=&vid=USPAT4973139&id=j9UkAAAAEBAJ&oi=fnd&dq=Automotive+Head+Up+Displays&printsec=abstract

[18] Chen, D.-y., G.-r. Chen, Y.-w. Wang, J.-w. Hsieh, and C.-h. Chuang (2012) "Real-Time Dynamic Vehicle Detection on Resource-Limited Mobile Platform," , pp. 15–17.

[19] Fang, W., Y. Zhao, Y. Yuan, and K. Liu (2011) "Real-Time Multiple Vehicles Tracking with Occlusion Handling," *2011 Sixth International Conference on Image and Graphics*, pp. 667–672.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6005949

[20] Wang, C.-c. R. and J.-j. J. Lien (2008) "Features A Statistical Approach," **9**(1), pp. 83–96.

[21] Yang, C., R. Duraiswami, and L. Davis (2005) "Fast multiple object tracking via a hierarchical particle filter," *Computer Vision, 2005. ICCV ...*, pp. 212–219 Vol. 1.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1541259http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1541259

[22] Kanhere, N. (2005) "Vehicle Segmentation and Tracking from a Low-Angle Off-Axis Camera," *Computer Vision and ...*.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1467573

[23] Maldonado-bascón, S., S. Lafuente-arroyo, P. Gil-jiménez, H. Gómez-moreno, and F. López-ferreras (2007) "Road-Sign Detection and Recognition Based on Support Vector Machines," **8**(2), pp. 264–278.

[24] Tsai, R. (1987) "A versatile camera calibration technique for high-accuracy 3D machine vision metrology using off-the-shelf TV cameras and lenses," *Robotics and Automation, IEEE Journal of*, (4).
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1087109

[25] Ponce, J. and D. Forsyth (2002) *Computer vision: a modern approach*, 1st ed., "Prentice Hall Professional Technical Reference".
URL http://onlinelibrary.wiley.com/doi/10.1002/cbdv.200490137/abstracthttp://www.inria.fr/centre/paris-rocquencourt/actualites/computer-vision-a-modern-approach

[26] Zhang, Z. (1999) "Flexible camera calibration by viewing a plane from unknown orientations," *Computer Vision, 1999. The Proceedings of the ...*, **00**(c), pp. 0–7.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=791289

[27] Hartley, R. and A. Zisserman (2004) *Multiple View Geometry in Computer Vision*, 2nd ed., Cambridge University Press.
URL http://onlinelibrary.wiley.com/doi/10.1002/cbdv.200490137/abstracthttp://journals.cambridge.org/production/action/cjoGetFulltext?fulltextid=289189

[28] NAL Research (1999) *Datum Transformations of GPS Positions Application Note, Tech. rep.*

[29] Gazebo (2013), "Gazebo Wiki," .
URL http://gazebosim.org/wiki

[30] Bouguet, J. (2001) "Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm," *Intel Corporation*, **1**(2), pp. 1–9.
URL http://robots.stanford.edu/cs223b04/algo_affine_tracking.pdf

[31] Bay, H., A. Ess, T. Tuytelaars, and L. Van Gool (2008) "Speeded-Up Robust Features (SURF)," *Computer Vision and Image Understanding*, **110**(3), pp. 346–359.
URL http://linkinghub.elsevier.com/retrieve/pii/S1077314207001555

[32] Lucas, B. and T. Kanade (1981) "An Iterative Image Registration Technique with an Application to Stereo Vision," *Proceedings of the 7th international joint ...*, pp. 674–670.

URL http://www.ri.cmu.edu/pub_files/pub3/lucas_bruce_d_1981_1/lucas_bruce_d_1981_1.ps.gz

Vita

Robert Leary

Address:
909 W. Aaron Dr.
State College, Pennsylvania 16803

Education:
B.S. in Mechanical Engineering with Honors, *2013*
The Pennsylvania State University, University Park, Pennsylvania

Honors and Awards:
Outstanding Mechanical and Nuclear Engineering Senior, *2013*,
The Pennsylvania State University, University Park, Pennsylvania

Boscov Academic Excellence Award, *2011*,
The Pennsylvania State University (Berks Campus), Reading, Pennsylvania

Berks Campus Honors Program Award, *2010-2011*,
The Pennsylvania State University (Berks Campus), Reading, Pennsylvania
Bangkok, Thailand