3-26-2020

Object Detection with Deep Learning to Accelerate Pose Estimation for Automated Aerial Refueling

Andrew T. Lee

OBJECT DETECTION WITH DEEP LEARNING TO ACCELERATE POSE ESTIMATION FOR AUTOMATED AERIAL REFUELING

THESIS

Andrew T Lee, 2d Lt, USAF

AFIT-ENG-MS-20-M-035

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-20-M-035

Object Detection with Deep Learning to Accelerate Pose Estimation for Automated Aerial Refueling

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Science

Andrew T Lee, B.S.Co.E., B.S.C.S.

2d Lt, USAF

March 26, 2020

AFIT-ENG-MS-20-M-035

Object Detection with Deep Learning to Accelerate Pose Estimation for Automated Aerial Refueling

THESIS

Andrew T Lee, B.S.Co.E., B.S.C.S.
2d Lt, USAF

Committee Membership:

Scott L Nykl, Ph.D
Chair

Clark N Taylor, Ph.D
Member

Brett J Borghetti, Ph.D
Member

Abstract

Remotely piloted aircraft (RPAs) cannot currently refuel during flight because the latency between the pilot and the aircraft is too great to safely perform aerial refueling maneuvers. However, an automated aerial refueling (AAR) system removes this limitation by allowing the tanker to directly control the RPA. The tanker quickly finding the relative position and orientation (pose) of the approaching aircraft is the first step to create an AAR system. Previous work at Air Force Institute of Technology (AFIT) demonstrates that stereo camera systems provide robust pose estimation capability. This thesis first extends that work by examining the effects of the cameras' resolution on the quality of pose estimation. Next, it demonstrates a deep learning approach to accelerate the pose estimation process. The results show that this pose estimation process is precise and fast enough to safely perform AAR.

Acknowledgements

I would like to thank my advisor, Dr. Nykl, for continuously going above and beyond in every aspect of supporting this research and my academic development. I would also like to thank my committee members, Dr. Taylor and Dr. Borghetti, for providing consistent input and valuable feedback throughout this process. Thank you to my family for their support, and my dog Skippy for always greeting me with a smile no matter how late I had to stay at work. Thank you to the Lair Boys for making working at AFIT a blast the entire time. Finally, thank you to the Dinner Squad, who made every Taco Tuesday one with stimulating conversation, close camaraderie, and a lot of fun.

This work is dedicated to my late father, who played a key role in making me the person I am today.

Andrew T Lee

Table of Contents

Page

List of Figures

List of Tables

Object Detection with Deep Learning to Accelerate Pose Estimation for Automated Aerial Refueling

I. Introduction

Automated aerial refueling (AAR) is an important emerging technology for the United States Air Force (USAF). Currently, remotely piloted aircraft (RPAs) cannot perform in-air refueling maneuvers due to the multi-second latency between the pilot and the remote aircraft. An AAR system bypasses this limitation by allowing the tanker to automatically control the receiver during the refueling process. The ability to refuel RPAs in-flight would provide a valuable increase in operational endurance for their mission set. Additionally, an AAR system serves as an important intermediate step to a fully autonomous tanker. An AAR system needs the capability to control the tanker, its refueling boom, and the receiver for the duration of the aerial refueling procedure. Before control logic can be safely implemented, the system must have high precision relative position and orientation (pose) estimation process that tracks the receiver in real time. To do this, the tanker requires a vision system that meets AAR system requirements.

The greatest challenge for an AAR computer vision system is difficulty in depth estimation caused by the long range from the cameras on a tanker to the refueling contact point. The contact point is approximately $30m$ from the stereo cameras. At contact, the 3D pose estimation error must be less than $10cm$. Due to aerial refueling mission constraints, existing aircraft cannot be modified and Global Positioning System (GPS) signals cannot be used to augment a vision-based approach. AFIT researchers have previously proposed a method for solving this problem using a stereo

vision system [5]. This thesis seeks to validate and improve that system.

One method for improving the accuracy of pose estimates is to increase the stereo images' resolution. Unfortunately, an increase in pixel count leads to a proportional increase in processing time.[1] To mitigate this limitation, this work draws inspiration from nature. The neural structure dedicated to processing imagery is too small to support high-resolution sensory over a wide field of view. Therefore, the biological eyes in predators have small, ultra-high-resolution fovea and a low-resolution periphery. This natural, ultra-high-resolution stereo vision system allows precise ranging and pose estimation in predators' brains without requiring more neural processing power. To replicate this nature-based approach, this work uses a convolutional neural network (CNN) to localize the receiver in each image and crop the images to only contain a tightly-bound rectangle with the receiver inside it. Then, the more computationally expensive image processing steps need only be applied to the region containing the receiver.

1.1 Problem Statement and Research Objectives

In its current state, pose estimation presents the largest limiting factor for AAR. Finding the precise relative pose of the receiver and then controlling it precisely enough to connect with the tanker is difficult. This thesis aims to answer two questions:

1. Can higher resolution stereo camera systems provide the pose estimation accuracy that AAR requires?

2. Can a machine learning approach speed this process up enough to use it in a real-time system?

[1] The complexity of the algorithm used to generate a 3D point cloud is $O(n)$ where n is the total number of pixels in a stereo image pair.

To answer these questions, this research augments the AAR vision pipeline proposed by Parsons et. al. in [5] by adding a CNN that autonomously detects the region of interest containing the receiver in the stereo image pair. This research seeks to demonstrate the following:

1. Higher resolution camera systems produce better pose estimates in simulations and real-world tests.

2. A CNN augmentation decreases pose estimation time.

3. A CNN augmentation does not decrease the accuracy of a given camera system's pose estimation.

Combined, these goals lead to an improvement of the AAR vision pipeline that creates a better solution to meet operational constraints for the Air Force.

1.2 Assumptions

To accurately model aerial refueling scenarios, this research assumes the receiver approaches on a linear flight path. A linear flight path is much easier to model in a ground experiment. Next, this research assumes the contact point is $30m$ from the stereo baseline. While this is not an exact measurement, it is close to the actual value, and allows an easy location for direct comparisons between systems and experiments. The region of greatest interest is when the receiver is $50m$ to $30m$ away from the stereo baseline. This research assumes the orientation variation is very small in a refueling approach, since a variation of more than one degree would quickly take the receiver out of a refueling approach. For this reason, the error analysis focuses exclusively on the 3D offset vector instead of a full position and orientation solution. This research assumes adequate lighting for the electro-optical (EO) cameras to function properly

and no occlusion from clouds or other aerial debris; it does not seek to quantify exactly how much light is required, and all experiments are performed in daylight.

1.3 Document Overview

Chapter II further explores previous efforts in AAR and computer vision. Chapter III details the design and implementation of the real-time computer vision pipeline for AAR. It then explains the design and integration of the image segmentation CNN. Chapter IV discusses the accuracy improvement in a real-world ground test from using 4K+ resolution cameras and the performance benefits of a deep learning augmentation by showing the speedup in simulations. Finally, Chapter V discusses the benefits that this process can yield in computer vision challenges facing the Air Force.

II. Background and Literature Review

This chapter outlines the problem domain for this thesis and surveys related work. Section 2.1 overviews aerial refueling and outlines constraints that guide this work. Section 2.2 explains the pinhole camera model, which is essential to computer vision. Section 2.3 describes how to calibrate a real camera to fit the pinhole camera model. Section 2.4 discusses how images are processed to produce depth maps using epipolar geometry and stereo block matching. Section 2.5 examines deep learning's role in computer vision. Section 2.7 discusses the use of simulations in the automated aerial refueling (AAR) domain. Section 2.8 examines the theory behind using better cameras to perform long range pose estimation. Finally, Section 2.9 explains how this work continues previous AFIT AAR work.

2.1 Aerial Refueling

Aerial refueling is the process of transferring fuel from one aircraft to another in-flight. A tanker aircraft transfers fuel to a receiver aircraft. There are two primary ways to perform aerial refueling: the boom method and the probe-and-drogue method. This research focuses on the boom method employed by the United States Air Force (USAF) as shown in Figure 1. In the boom method, the receiver approaches the tanker from behind and below. The approach maneuver begins when the receiver is approximately $\frac{1}{2}km$ from the tanker. The approach ends when the receiver reaches the contact position. For this research, contact is approximated as $30m$ from the tanker's camera system. At the contact position, a boom operator injects the boom into a receptacle on the receiver and begins refueling. The space the receiver occupies during refueling is called the refueling envelope. The following constraints guide this research:

- AAR should not depend on precision GPS information because GPS is jammable and spoofable.
- Receiver aircraft cannot be modified with sensors or markers to aid the pose estimation process. While adding sensors or markers to the receiver would provide better results, these solutions are impractical and could adversely affect receiver performance.
- The navigation solution must run in real time. A system that takes too long to calculate its navigation solution cannot safely control the refueling procedure.
- The boom method requires precise control of the tanker, boom, and receiver from the pilots of both aircraft and the boom operator. Aircraft that cannot perform these movements within minimal fault tolerances cannot refuel in-air.
- Pose estimation is the most difficult component of the process in almost all cases. To safely control the receiver, the tanker must know the exact relative position and orientation of the receiver. Inaccurate pose estimation risks putting the receiver on an incorrect flight path. This could lead to failure to connect and damage to both aircraft.

To safely and properly control the receiver, the tanker must know the exact relative position and orientation of the receiver. Inaccurate pose estimation risks putting the receiver on an incorrect flight path. This could lead to failure to connect and damage to both aircraft. Several efforts have been made to perform AAR. One effort to accomplish pose estimation added markers to the tanker and receiver, next using geometric algorithms to calculate pose [6]. Another approach used precision GPS and inertial measurement unit (IMU) with Kalman Filtering to estimate pose [7]. These solutions do not meet mission constraints outlined previously: specifically, this

work seeks to have a real-time, vision-only approach pose estimation at ranges from $30m - 50m$.

Figure 1: F-35 connects with KC-46 Tanker using the boom method [1]

The latency from remote control of remotely piloted aircraft (RPAs) makes AAR maneuvers unsafe [8]. AAR would give RPAs the ability to refuel in-air, because a tanker that can control the receiver would bypass the latency problem. Allowing RPAs to refuel in-air would have several benefits. The aircraft could remain on station virtually indefinitely, limited only by maintenance requirements or munitions load. Crew fatigue could be mitigated in RPAs by changing the crew during flight. The ability to autonomously connect to receivers could serve as a stepping stone towards a fully autonomous tanker aircraft, which could help alleviate pressures on pilot supply.

2.2 Pinhole Camera Model

The pinhole camera model applies to most conventional cameras. A set of digital sensors creates a 2D image from light focused through a lens. Each sensor creates a pixel, and the pixels correspond to spatial properties based on camera's physical

properties. The pinhole model works because it models how light passes through the camera's aperture [9]. Figure 2 shows how the light from an object at point p passes through pixel (u, v). However, any point on the line extending from F_c through P and onward could also be represented in pixel (u, v). At a higher resolution, (u, v) will correspond to a smaller volume of space and can help derive a more precise 3D re-projection.

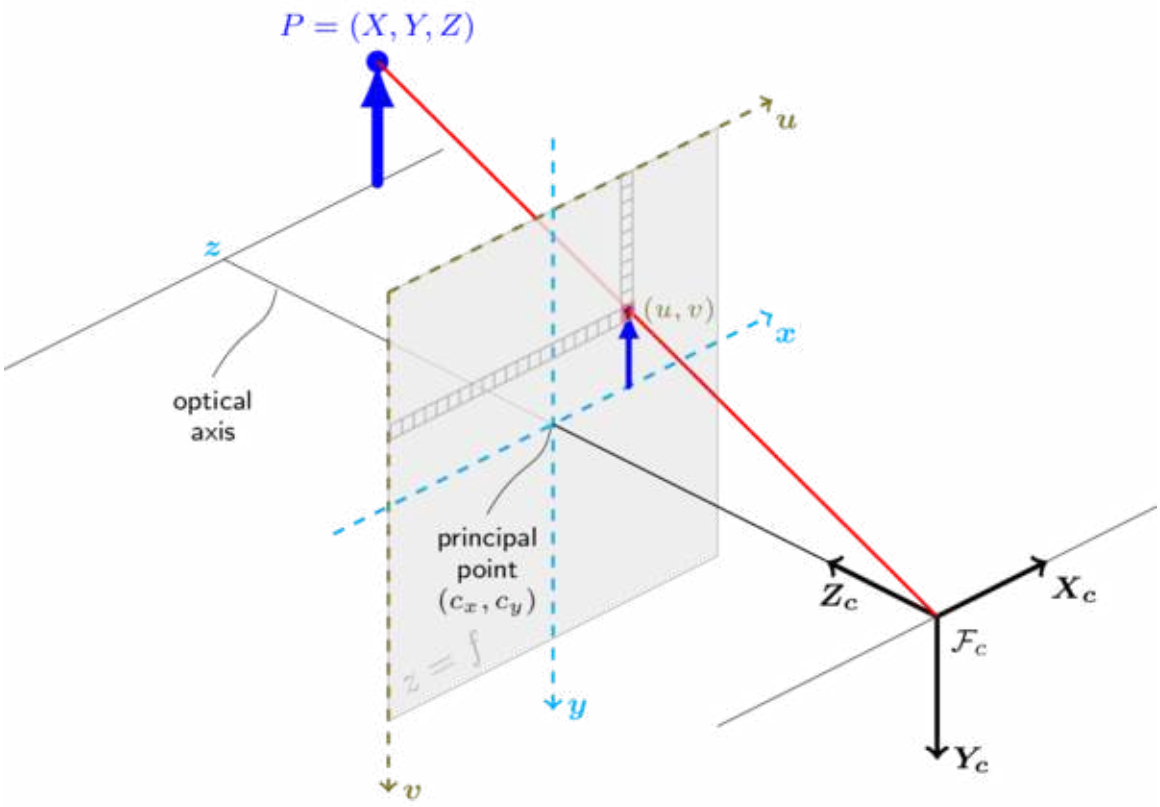


Figure 2: Pinhole Camera Model [2]

2.3 Camera Calibration

Camera calibration allows real cameras to more accurately represent the pinhole camera model. Physical lenses and sensors introduce small distortions and translations that prevent direct derivation of 3D information from the 2D image. To allow this transformation, camera calibration parameters are used to un-distort and rectify

the stereo images. A stereo camera system that is undistorted and rectified can then function as if the two cameras were ideal pinhole cameras. Rectifying stereo image pairs requires knowledge about the intrinsic characteristics of each camera and the relative pose of the cameras. The intrinsic camera calibration is denoted as M_1 for the left camera and M_2 for the right camera. These matrices take the form

$$M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

where f is the camera's focal length in pixels for x and y, and c is the center-pixel coordinate in x and y. Additionally, the relative pose of the cameras needs to be determined. This pose takes the form of a 3×4 matrix

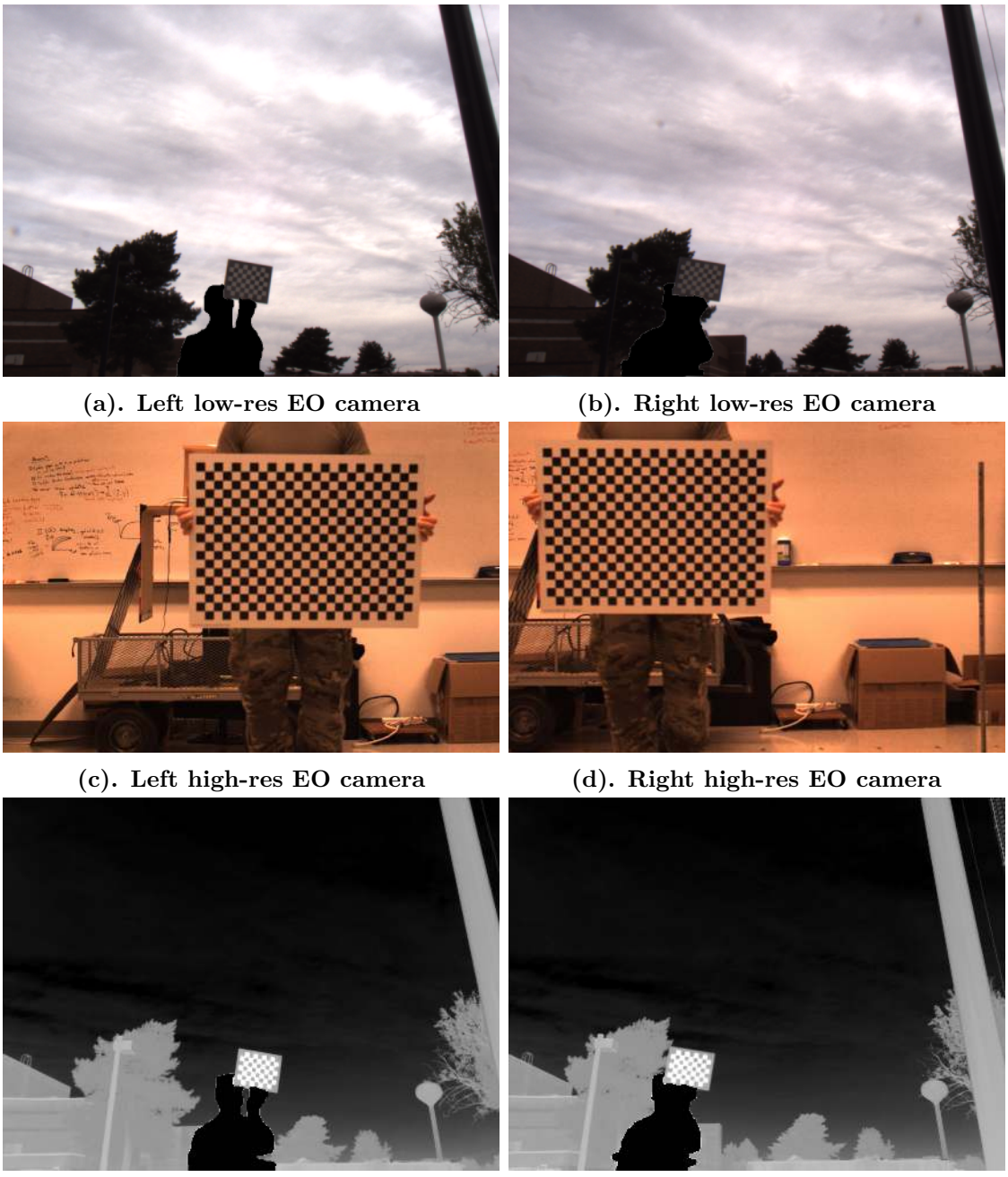
$$[R|t] = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix}$$

where R is the direction cosine matrix (DCM) and t is the vector from the left camera's principle point to the right camera's principle point. Knowing these parameters, it is possible to create rectification maps that allows a computer to remap images into rectified pairs in real time.

Images must also be undistorted before rectification; generally the distortion is modeled using a polynomial with coefficients denoted $k_1 ... k_n$. Distortions for high-end cameras tend to be small.

Camera calibration can be solved manually; however, the most common practice is to take images of a checkerboard of known dimensions at different poses and then allow a computer vision library (such as OpenCV) function to solve for the necessary parameters. Figure 3 shows an example calibration image pair for each set of cameras used in this research. In theory, it is possible to calibrate off of one image, but in reality, ten to twenty good image pairs are required.

Camera calibration is difficult: there are 24 correlated parameters. Some have linear effects, and others have nonlinear effects on 3D re-projection. For example, on a given image, it may be difficult to determine if a re-projection error is properly minimized by a narrow stereo baseline and the cameras facing forward or a wider stereo baseline where the cameras are rotated in. Small variations in rotation add more error the farther a target feature is from the stereo baseline. Currently, OpenCV struggles to make these small distinctions which can lead to large errors at longer ranges.



(a). Left low-res EO camera **(b). Right low-res EO camera**

(c). Left high-res EO camera **(d). Right high-res EO camera**

(e). Left IR Camera **(f). Right IR Camera**

Figure 3: Calibration Images for EO and IR Cameras, low-res images from [3]

2.4 Epipolar Geometry and Stereo Block Matching

The pinhole camera model allows pixels to be re-projected into 3D space if relative orientation of the cameras is known. Because real cameras are not pinhole cameras, the images must first be undistorted and rectified. Next, 3D information is derived using epipolar geometry. Figure 4 shows how the location of pixels X_L and X_R can be used to triangulate the position of point X given e_L, e_R, O_L, and O_R [10].

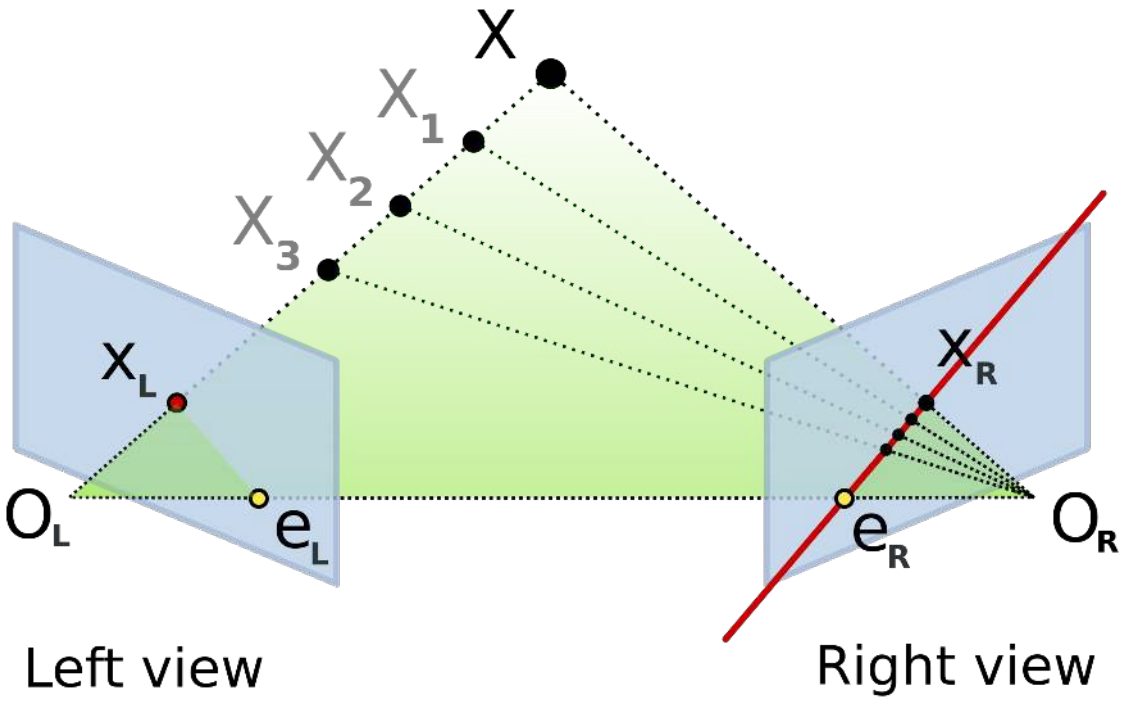


Figure 4: Epipolar Geometry [4]

The line $e_L \rightarrow e_R$ is an epipolar line. A pair of images where all of the epipolar lines are parallel is called a rectified image pair. Rectification allows much faster feature searches in the image pair because a feature must be on the same line of pixels in a rectified image pair; no such guarantee exists for a raw image pair.

Stereo block matching allows the transform of rectified image pairs into depth maps based on features in both images. If the disparity (distance in pixel-space

coordinates) is larger, the object is closer. Conversely, a smaller disparity denotes a more distant feature. A disparity of zero occurs when the feature is so distant that the same pixel in both cameras finds it. Figure 5 shows how the stereo block matching algorithm finds the tree top in both images. Once it is found, the disparity allows the distance to the tree to be calculated. When this process is completed on the entire scene that is visible to both cameras, it creates a depth map. This depth map can be reprojected into 3D space to create a point cloud of the environment.

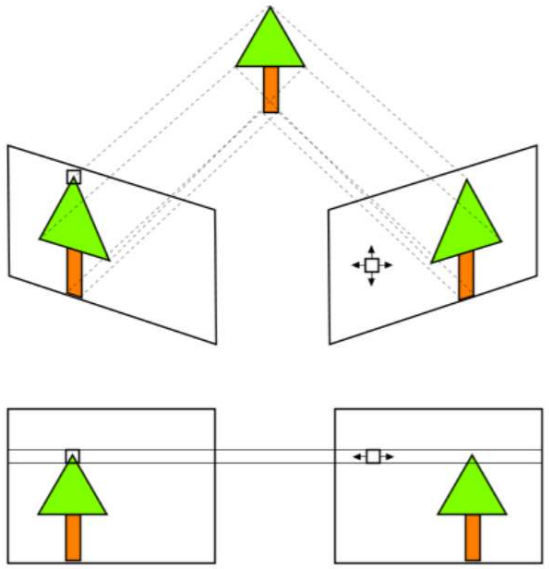


Figure 5: Rectified stereo images. This figure shows how a sliding window can perform stereo block matching. In this case, the feature being found is the top of the tree.

One clear limitation of stereo block matching is computational intensity. The computational cost comes from the total number of pixels in the stereo image pair: the algorithm's complexity is $O(n)$ where n is the number of pixels. For example, if the resolution for each camera in a stereo image system is doubled, the total number of pixels is eight times greater, and the algorithm will require eight times more computation to complete.

2.5 Deep Learning in Computer Vision

Deep learning provides different trade-offs and benefits for real-time computer vision than conventional methods like stereo block matching [9]. Convolutional neural networks (CNNs) require training time, which depends on the hardware used, the size and structure of the network, and the training data. Training times thus vary greatly, and range from several minutes to several weeks. CNNs also require large, labeled training datasets. However, on-line a CNN will execute in a short, constant time for each input. The execution time is a function of the size of the model and the computer system running the model. As [11] discusses, hybridization of techniques may provide robust solutions.

Deep learning techniques have made substantial progress in recent years towards recognizing various objects in images. Since 2010, error rates in detecting objects in an image of a large dataset has decreased from over 20% down to 1%. Architectures vary from a sequential series of convolution filters all the way to dense connections where each layer takes in the output of every previous layer as input [12; 13]. Additionally, many newer CNNs have demonstrated an ability to localize the objects they identify in an image [14; 15; 16]. There have been experiments that show deep learning can outperform traditional methods [17]; however, conventional computer vision algorithms' intermediate representations, such as point clouds, have been shown to improve performance of deep learning solutions compared to image-only networks [18; 19].

In this work, a CNN is trained and tested using simulated imagery. Recent literature suggests that a network capable of performing well on the simulated imagery could be trained to perform as well on imagery from physical cameras. The work in [20; 21] suggests that the main issue is that the virtual camera is a different sensor from a physical camera, and domain adaptation is required when changing sensors.

Goyal et. al. [22] also show that augmenting semantic segmentation image datasets with synthetic imagery can improve results. Ros et. al. [23] demonstrate the benefits of this approach. CNN architectures that demonstrate capability when trained and evaluated on virtual imagery tend to perform just as well when trained and evaluated on physical imagery.

2.6 6DoF Pose Estimation

The primary goal in many computer vision systems is to derive a 3D model of the environment from 2D images. In monocular vision, information comes from a single camera or image. Zhang et. al. [24] demonstrate a deep learning process that performs object detection and pose estimation from a single camera. While their approach does run in real time, their neural network performs pose estimation on small objects that are very close to the camera. Similarly, Ferrara et. al. [25] use monocular and stereo systems to perform pose registration at ranges from $0.5m - 4.0m$. These approaches are not adequate solutions, because our problem requires a high precision solution for a very distant, large object.

Early AAR pose-estimation efforts focused on using monocular vision as a single component of a sensor-fusion relative navigation solution [6; 26]. Since both of these added markers to the aircraft, they do not meet the first or second constraints outlined in Section 2.1.

Stereo vision finds features in images and, after a calibration and rectification process, re-projects these features into 3D space relative to the cameras using epipolar geometry (for more information, see [10]). The stereo block matching algorithm locates features in both images and calculates the disparity, or distance in pixel-space, between them. Once disparities have been calculated for an image pair, the disparity map can be converted into a depth map. Stereo block matching requires a series of

pixel-wise comparisons. Increasing the number of pixels in the image pair leads to a linear increase in computation time. Our real-time constraint imposes a limit on the resolution that can be used in for this process. Once a point cloud has been generated, there are many techniques to perform pose estimation. In [27], Tam et. al. survey several registration methods. Since aircraft are rigid bodies, point-to-point iterative closest point (ICP) [28] was chosen for this work. However, alternate methods such as parallel ICP [29], fast global registration (FGR) [30], or a deep learning approach [31] may provide different performance and precision trade-offs.

2.7 3D Virtual World

Flight tests can be prohibitively expensive and take long planning periods. Moreover, truth data can be difficult to obtain. Simulation in a 3D virtual world can mitigate these problems and allow rapid prototyping and development. Campa et. al. [32] created a 3D virtual world to simulate aerial refueling approaches in 2009. Similarly, Parsons et. al. [5] created a more modern 3D virtual world that models the AAR environment. Fravolini et. al. [26] also used a 3D virtual world for their tests. In short, simulations are standard for this type of work.

In general, a computer vision algorithm's results in a 3D virtual world can be directly compared with a different algorithm's results in the same simulated environment. There is ongoing work to determine how well certain 3D virtual worlds correspond to physical experiments [33].

2.8 Camera Resolution and Depth Estimation

Intuitively, one expects that a higher resolution camera pair would improve depth estimation fidelity. In this subsection, we simulate the error in depth reprojection for a single point using OpenCV. With properly calibrated cameras, it

is possible to have a mean depth estimation error near zero at long ranges; however, the mean error is often misleading because individual depth estimations may significantly over-estimate or under-estimate an individual feature's depth. As outlined in Section 2.6, the entire point cloud contributes to the accuracy of pose estimation. For this reason, we seek to decrease mean absolute error (MAE). To demonstrate the necessity for using higher-resolution camera, we used the scenario demonstrated in Figure 6, where the point being triangulated was $30m$ away from a stereo camera system employing a $\frac{1}{2}m$ stereo baseline. Using cameras with a fixed, 56° field of view, the cameras' resolution was varied and compared the average error in depth estimation as a function of distance from the camera baseline. With Gaussian noise and a 1-pixel standard deviation in both images, a 1280×960 image resulted in $0.4598m$ MAE in distance from the cameras. By using a higher resolution camera of 4896×3264, a $0.38m$ MAE is achieved. This demonstrates the potential for significantly improving the relative pose computation of a stereo vision system by increasing the resolution of the cameras. These results correspond well to the error equation $\epsilon_z = \frac{z^2}{bf}\dot{\epsilon}_d$ [34], where ϵ_z is the depth error, z is the depth, b is the baseline, f is the focal length (in pixels), and ϵ_d is the matching error in pixels (disparity values, assumed to be one).

Table 1: Estimated depth for a feature located $30m$ away from stereo cameras

Resolution	**Mean Depth Estimation**	**MAE**	**Calculated Error**
1280×960	$30.10m$	$1.427m$	$1.496m$
1920×1440	$30.02m$	$0.9435m$	$0.997m$
3840×2880	$30.00m$	$0.4650m$	$0.498m$
4896×3264	$30.00m$	0.3844	$0.3910m$

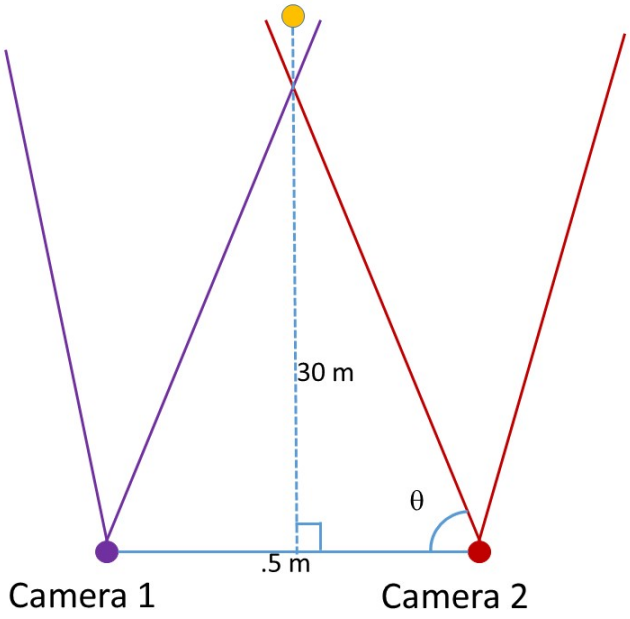


Figure 6: scenario for single-point depth estimation

2.9 Previous AFIT Work

This work follows Dallmann's thesis work [3]. Dallmann conducted a ground experiment to evaluate the capability of the pipeline designed by Parsons et. al. [5] to provide pose estimation for AAR. Analysis of his results shows that pose estimation errors at the contact point (assumed to be $30m$) for a 1280×960-resolution stereo electro-optical (EO) camera system are too high to safely control a receiver. This work seeks to improve these processes to meet precision and timing requirements for AAR. While Dallmann's results are presented in Chapter IV to provide context for the high-resolution vision system this work examines, it is important to discuss his results here as well, since these previously represent the state-of-the-art in near-real-time pose estimation. In the ground experiment, a pseudo-receiver is pushed towards a pseudo-tanker carrying a stereo vision system (Chapter III fully details the experimental methodology). Figure 7 shows the path estimation error as a function of distance for the EO camera system in his experiment. To safely perform AAR, the

3D error[1] would need to be below $10cm$ at a range of $30m$.

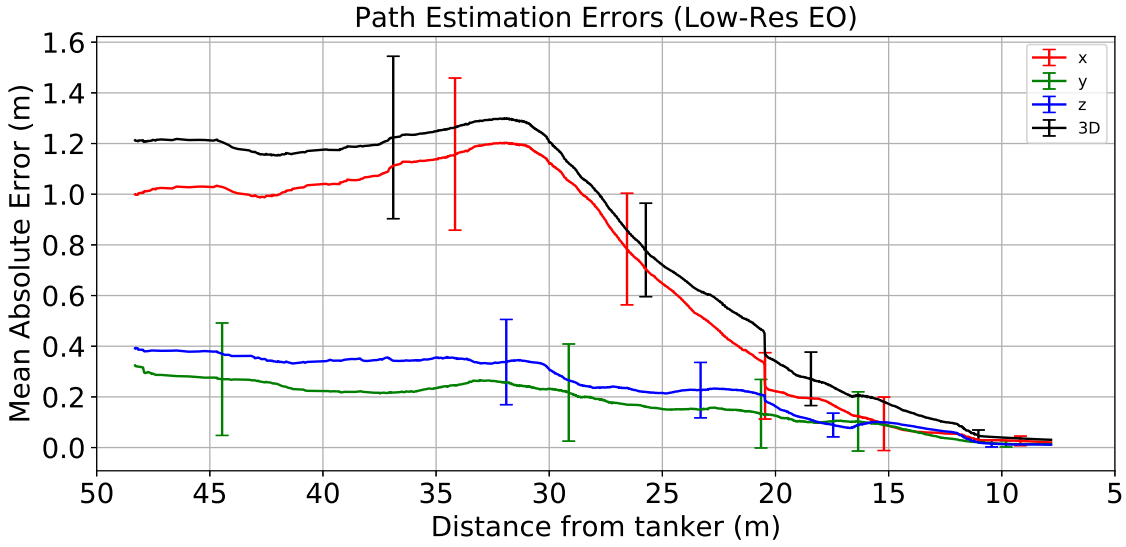


Figure 7: scenario for single-point depth estimation

Similarly, Figure 8 shows the infrared (IR) camera system's performance. Again, the errors are substantially larger than the requirements for AAR. This thesis work implements a system that meets the precision requirements but runs in approximately the same amount of time as this benchmark.

[1]The 3D error is the Euclidean distance between the *truth position* and the *sensed position*.

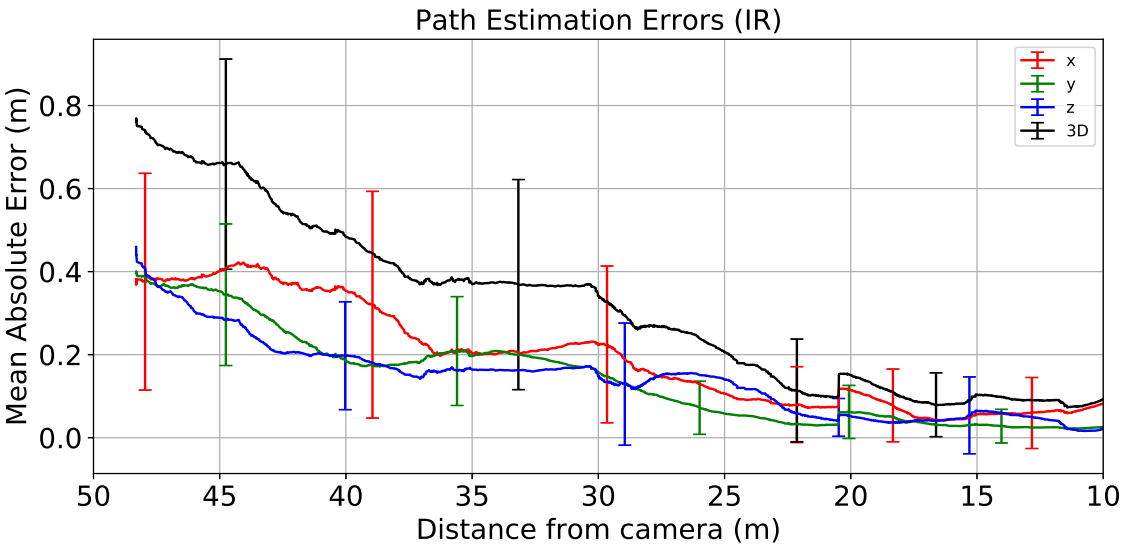


Figure 8: scenario for single-point depth estimation

III. Methodology

This chapter explains the experiments that were conducted to validate the proposed computer vision pipeline. The base pipeline comes from [5]. First, Dallmann's ground experiment is re-created using higher resolution camera system [3]. Next, a deep learning model is leveraged in the 3D virtual world to accelerate the pose estimation process. The model crops the stereo images to only include a tightly-bound rectangular portion containing the receiver, thus significantly fewer pixels need to be processed using the stereo block matching algorithm. This results in a significant speedup.

Section 3.1 describes the ground experiment that was recreated to show the affects of higher resolution cameras on the stereo vision pipeline's effective range. Section 3.2 describes how the deep learning model was designed and implemented. Finally, Section 3.3 explains how the convolutional neural network (CNN) was fully integrated into the vision pipeline.

3.1 Ground Experiment

The ground experiment seeks to validate the computer vision pipeline from Parsons et. al. [5]. The pipeline is as follows:

1. Capture stereo imagery
2. Generate a disparity map
3. Convert the disparity map into a 3D point cloud
4. Use iterative closest point (ICP) to register the receiver's pose

This section examines the affects of camera resolution on long range pose estimation using a ground experiment designed to mimic an aerial refueling approach.

Ongoing work suggests that this experiment's residual errors at a given range closely reflect a real test flight's residual errors at the same range [33]. The ground test used to compare the high-resolution cameras with lower resolution cameras was designed specifically to mimic an aerial refueling approach as much as possible. The experiment was run in two parts: the first with lower resolution electro-optical (EO) and infrared (IR) cameras [3], the second with high-resolution EO cameras.

3.1.1 Stereo Camera System

Two separate stereo vision systems comprised of two pairs of stereo EO cameras and one pair of IR cameras were employed. Using both EO and IR cameras increases the variability of the experiments and provides multiple data collection sources for analyzing. The use of IR cameras provides the opportunity to validate stereo IR cameras as a viable option for stereo vision in the AAR domain.

Allied Vision Proscilica GT1290C EO cameras were chosen for the low-resolution EO stereo vision system. The GT1290Cs capture 24-bit RGB images at a resolution of 1280×960 and have adjustable focal points and apertures. The adjustable focal point has the advantage of setting the focus to infinity to maximize image clarity for objects at long distances, since a receiver in the experiments is at a contact distance of about $30m$. Additionally, the cameras do not auto focus, which interferes with the camera calibration. For the high-resolution cameras, Allied Vision Prosilica GT4905C EO cameras were chosen. These have a compatible application programming interface (API) with the GT1290C, allowing for the same configuration except for the resolution. The cameras are capable of a full resolution of 4896×3264; however, to achieve $10Hz$ frame rates, the high-resolution cameras were configured to capture images at a resolution of 2448×1632 in a smaller field of view; they maintain 4K+ pixel-density if extended to the full field of view. The IR cameras chosen for the project had an

image resolution of 1024×768 and the images produced are 16-bit grayscale images. Like the EO cameras, the IR cameras were also focused to infinity. All three systems have similar full fields of view and aspect ratios.

Figure 9 shows the stereo camera configuration for the low-resolution EO cameras and the IR cameras. Figure 10 shows the mounted high-resolution cameras. The IR cameras can also be mounted with the high-resolution cameras as shown. The cameras were configured to trigger on a hardware signal controlled by the collection program. This ensures that each stereo image pair is captured at exactly the same time, and the pairs are timestamped for alignment with truth data. Images were collected at $10Hz$.

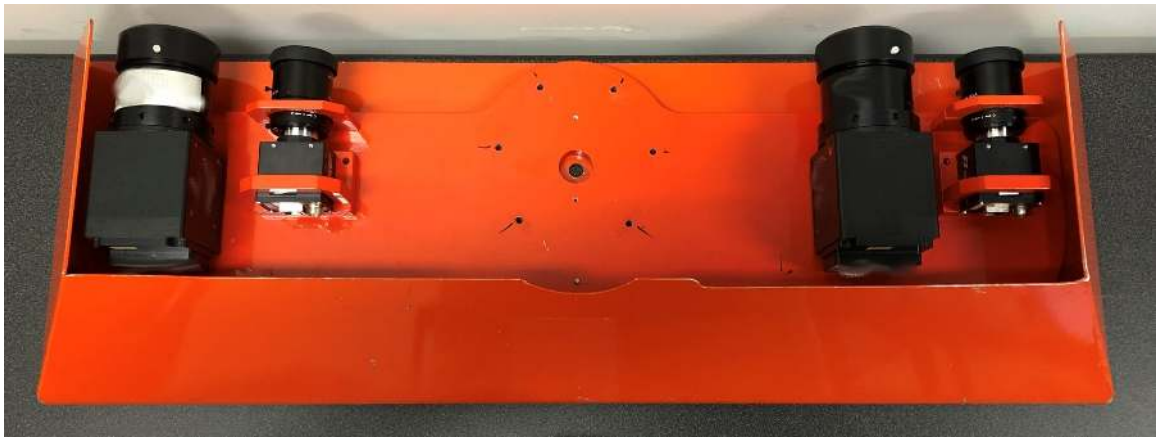


Figure 9: Low-resolution EO cameras and IR cameras mounted for the first experiment

3.1.2 Calibration

To perform image rectification and feature extraction, cameras must be calibrated properly. Dallmann used a metallic checkerboard with $30mm$ square tiles to capture calibration images for the low-resolution EO cameras and the IR cameras. The high-resolution EO cameras were calibrated using a larger, matte checkerboard. OpenCV's stereo calibration function was used to compute the calibration parame-

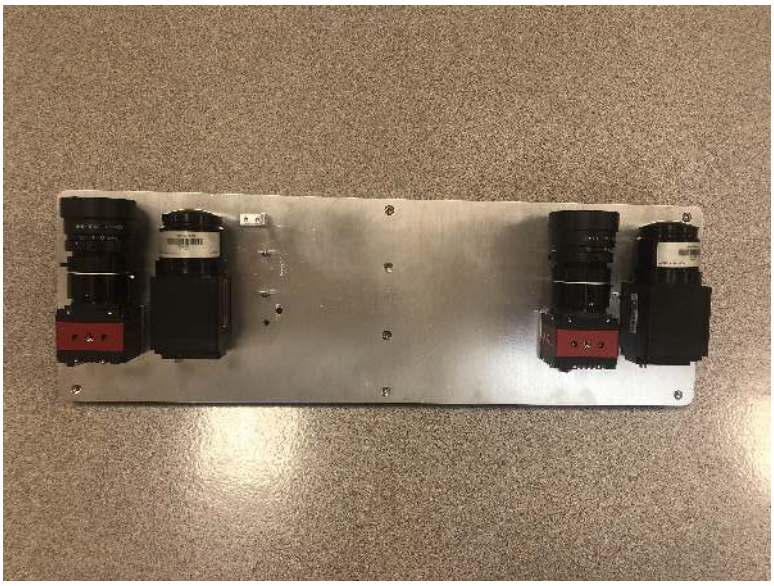


Figure 10: High-resolution EO cameras and IR cameras mounted for the first experiment

ters as described in [9]. The checkers on the metalic board were painted using white, heat-insulating paint. This creates a temperature differential that mimics the color differential, and the same board could be used to calibrate the EO and IR cameras, however, it is easier to calibrate accurately with the larger board.

3.1.3 Truth Data

To produce accurate position truth data for the psuedo receiver that is used during experiments, a differential GPS (DGPS) system creates a differential solution between a Primary system and Secondary system. The DGPS data is used to calculate the pose truth data. Changes in orientation more than one degree would quickly force the receiver out of the refueling envelope in an aerial refueling approach. Therefore, while ICP returns a 6DoF rigid body registration, the main concern is the 3D offset vector. The Primary and Secondary systems both collect and save their GPS data for post-processing. The DGPS system collects at at $5Hz$. To ensure that the pose is correlated to the correct stereo image pair, the computer's clock is synced with GPS

time using a Time Machine TM2000A time server, and each image is time-stamped. The two closest DGPS solutions are linearly interpolated to the time that the image pair was captured: this solution is used as the truth data for a given image pair. The DGPS system has centimeter-level error. An error this small would be sufficient for an aerial refueling connection using the tanker method. A system verified by this methodology would be directly deployable.

3.1.4 Pseudo-Tanker and Pseudo-Receiver

The pseudo tanker was designed using a wagon that could securely support the vision system, the data collection computer, the Primary DGPS system, and a power supply system. Additionally, the GPS antenna was placed above all of the equipment to prevent the blocking or multi-path GPS signals traveling to it. Figure 11a shows the pseudo tanker with the IMU axis attached.

The pseudo-receiver was designed to mimic the scaled-down behavior of a generic receiver in a refueling approach. The main structure is a wing and body with patterns printed on it. Figure 11b shows the front view of it. Patterns are placed on the surface to mimic the paint variations, rivets, and other surface features that stereo block matching can locate on the surface of an approaching aircraft.

For pose registration, a reference point cloud (red) is matched onto a sensed point cloud (yellow) using ICP [28]. Figure 12 shows the reference point cloud for the pseudo-receiver and an example of a sensed point cloud.

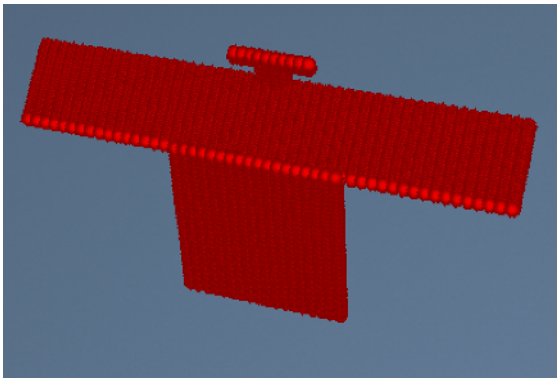
3.1.5 Running the Experiment

The experiment was conducted in a parking lot to allow a large, relatively flat, open space. The pseudo-tanker remained stationary, and the pseudo-receiver was pushed towards it. Several approaches were conducted.

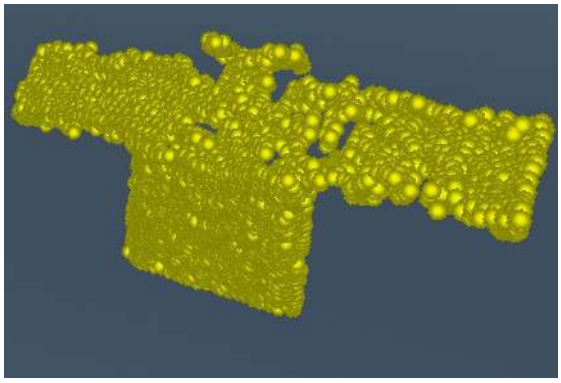
(a). Rear view of pseudo tanker cart with primary axes shown

(b). Front view of pseudo receiver cart

Figure 11



(a). Red Reference Point Cloud



(b). Yellow Sensed Point Cloud

Figure 12: Reference Model and Sensed Model as Point Clouds

After the tests were conducted and truth data was obtained from post-processing the DGPS data, the computer vision pipeline was applied to estimate the pseudo-receiver's pose. Figure 13 shows an example of registration being visualized in the virtual environment. This allows for recreation of the experiment and visualization of the pose estimation in post-processing. The data from [3]'s experiment was re-processed in this way to provide better comparison with the high-resolution camera system.

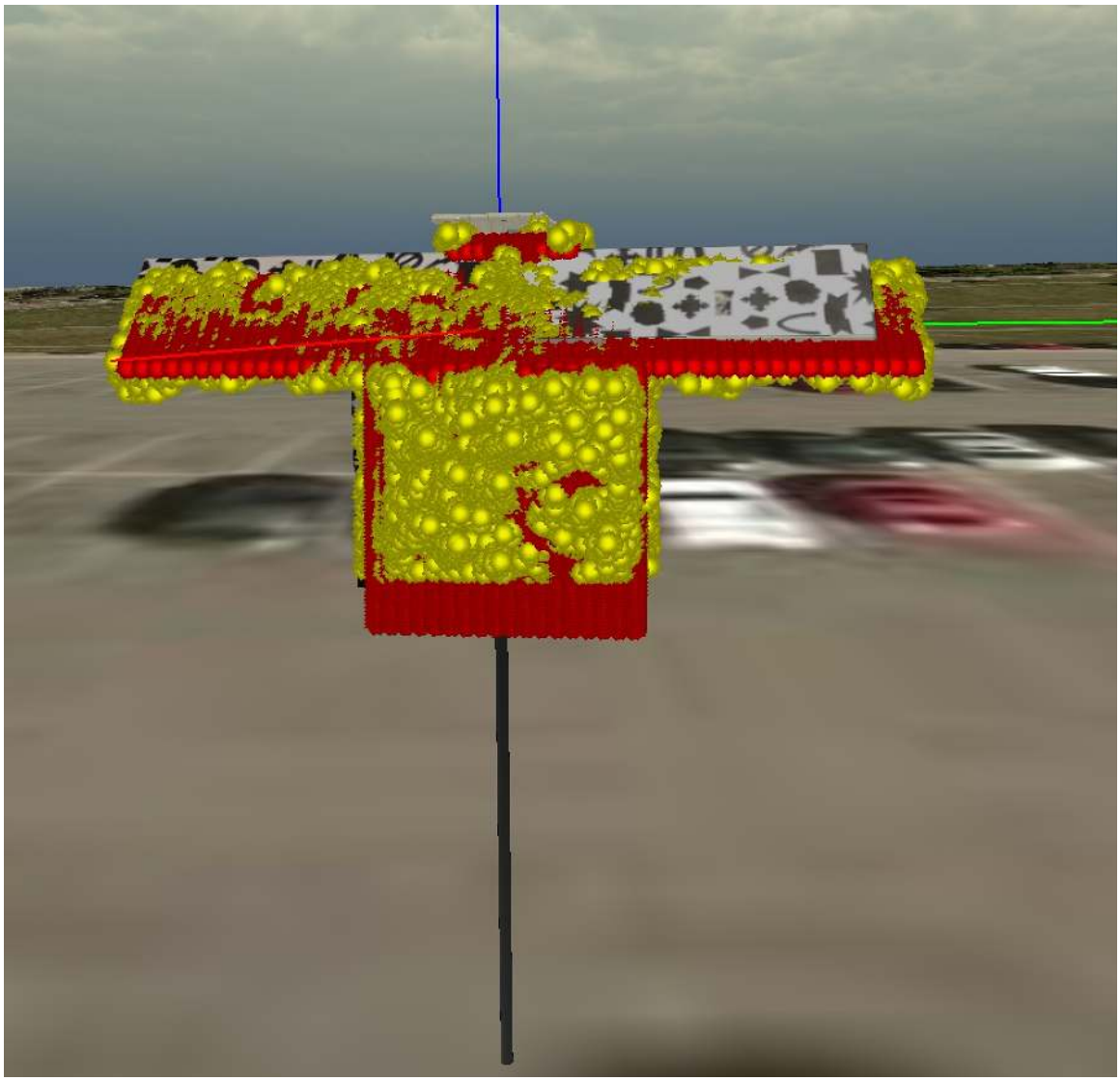


Figure 13: Registration of the Reference Point Cloud with the Sensed Point Cloud

3.2 Pipeline Augmentation with a CNN

To generate a 3D point cloud faster while using high-resolution imagery, the base vision pipeline (shown in Section 3.1) is modified by cropping the high-resolution images before generating a disparity map. This leads to an improved pipeline that this thesis now seeks to evaluate:

1. Capture high-resolution stereo imagery

2. Use CNN to dynamically crop the stereo images

3. Generate a disparity map only for the region of interest

4. Convert the disparity map into a 3D point cloud

5. Use ICP to register the receiver's pose

To perform the dynamic cropping, the vision pipeline is augmented with a deep learning model that is trained to segment computer-simulated imagery of a receiver aircraft. This model is deployed in a 3D virtual world refueling simulation. The model crops the stereo images to only include a tightly-bound rectangular portion of the original image containing the receiver; thus, significantly fewer pixels need to be processed using the stereo block matching algorithm. This results in a significant speedup without sacrificing precision. Section 3.2.1 describes the 3D virtual world. Section 3.2.2 details how our deep learning model was designed and tested.

3.2.1 Computer Simulation

To quantify performance benefits and simulation accuracy, simulations are performed in the AFTR Burner Engine [35]. The AFTR Burner Engine is a custom 3D graphics engine based on OpenGL that uses geometrically accurate models, high quality textures, and realistic lighting to replicate real refueling approaches. This is the same simulation environment that several researchers have used [5; 36; 37] for their automated aerial refueling (AAR) experiments. The cameras in the simulation have the same resolution and field of view as their physical counterparts used in the ground experiment. OpenGL's rendering pipeline introduces small variations and distortions due to the discretization of the pinhole camera model, so it is necessary to perform multi-sample anti-aliasing (MSAA) and camera calibration. The calibration is performed using a virtual checkerboard and OpenCV's stereo calibration function

as described by Kaehler and Bradski [9]. To verify that increased resolution improves pose estimation, a simulated refueling approach was conducted in the virtual world using cameras at different resolutions and the fidelity of the pose estimation is compared.

3.2.2 CNN Design

For this research, a basic deep CNN architecture was created. There are many existing architectures, such as YOLO [14], that could perform a similar function. The work here exists to demonstrate how a deep learning augmentation to improve a computer vision pipeline can be implemented, and even a basic model can yield large performance benefits. The remainder of this subsection explains how the network was trained and evaluated. Its image segmentation performance is then briefly discussed.

3.2.2.1 Data

The dataset for this project was created using the AFTR Burner engine described in Section 3.2.1. In the simulation, a virtual receiver was placed at random, uniformly distributed locations within the camera's field of view at distances between $20m$ and $100m$ and its orientation was randomly adjusted by small amounts to ensure diversity. One of several background images of real landscapes from aerial views was placed in the background. The engine used a virtual camera to capture a 1280×960 resolution image of the simulated scene. Next, the simulation was modified to reskin the receiver in a flat, distinct color. After a pair of images was captured, the background was changed and the receiver was moved. Figure 14 shows an example of an image pair. For this research, $5,500$ input/truth image pairs were generated. The project used $5,000$ pairs for training and validation and 500 pairs for testing. The test set was generated using different background images that were unseen in

training.

Figure 14: Example training image pair captured from AFTR Burner

To create the truth data, a mask was applied to the image to locate the pink skin on the receiver in the truth images. The minimum and maximum pixel coordinates were used to calculate the center x, center y, width, and height of the box in pixel space. These were saved in a CSV with the associated image number. Each of the training images was blurred using a 3×3 low-pass blurring filter to help prevent the model from over-fitting potential sharp edges in the simulated imagery.

The model was further designed to perform image segmentation specifically for AAR. High-resolution cameras can often capture at higher frame rates in grayscale than in color; however, full resolution is not necessary to localize the receiver. Down-sampling images allows for accurate localization with smaller networks. For these reasons, training images were converted to 512×386 grayscale. Brightness was varied in each image between 5% and 300% to help feature selection become less dependent on specific lighting. Pixel values were then rescaled to floats between 0 and 1. For testing, the pixel values are rescaled but not otherwise augmented.

3.2.2.2 Model

This research used a new deep CNN model. The model has 16 convolution layers and two fully connected layers before output. It takes an image as described in Section 3.2.2.1 and outputs regression values for the bounding box as center-x, center-y, width, and height values, normalized to be in range $[0-1]$ as a proportion of the original image. Figure 15 shows a high level view of the model.

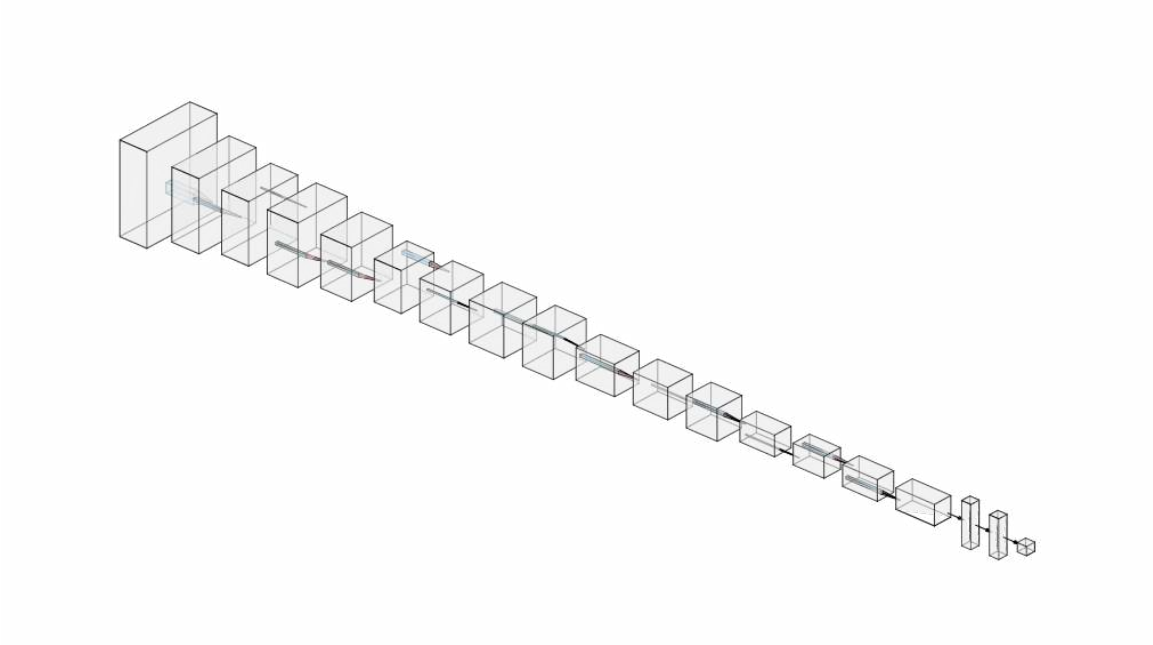


Figure 15: The CNN model used for this research

Batch normalization is performed at each layer. A leaky ReLU function[1] serves as the activation function for each layer. Leaky ReLU can prevent *dead nodes* that activation functions with finite ranges on an infinite (or very large) domain can create. For example, the sigmoid[2] function has a range $(0, 1)$ but is greater than $\sigma(x) > 0.99$ for $4.6 < x < \text{inf}$ and $\sigma(x) < 0.01$ for $-\text{inf} < x < -4.6$. It is likely that with

[1]
$$f(x) = \begin{cases} 0.1x & x \leq 0 \\ x & x > 0 \end{cases}$$

[2] $\sigma(x) = \frac{e^x}{e^x+1}$; the sigmoid function has fallen out of favor in newer architectures.

a range this big, the back-propagation process will cause some nodes to always be "on" or "off" regardless of input, which decreases the model's capacity. In contrast, leaky ReLU function ensures that all of the parameters associated with nodes will contribute to the model's solution.

There is a 40% dropout before each fully connected layer and a 20% dropout before the output layer. These regularization techniques help the model train faster and on smaller datasets, but do not affect the model's execution speed or capacity, and are evident in many deep learning architectures [14; 15; 12; 13].

The first convolution layer uses large 12×9 filters with a 4×3 stride. This takes in the $512 \times 386 \times 1$ images and outputs a square $128 \times 128 \times 128$ feature map. Using a large filter with a stride on the first layer is common practice [14; 15] and helps reduce the dimensionality of the initial image while finding features and having relatively few parameters. The remaining convolution layers vary between 3×3 convolutions, 1×1 convolutions, and periodic max pooling. The 3×3 convolution layers perform much of the feature extraction. The 1×1 convolution layers add nonlinearity (since they perform a linear transform and pass through the non-linear activation function) while adding fewer parameters than a larger filter would. Keeping the parameter count small makes the model train and execute more quickly. When the final pooling occurs, the feature map is $1 \times 1 \times 1024$, and is flattened to a one-dimensional vector of 1024 features. The two dense layers of 1024 nodes and 512 nodes perform regression from the feature map, and then the output layer predicts the bounding box. See Appendix 1.3 for information on the code base for the model.

The workstation training and evaluating the model had an Intel i7-7820X processor, 96GB of main memory, and an Nvidia 1080Ti GPU. The model trained in less than three hours. By functioning on a consumer-level computer, the model demonstrates that it can be used in practical settings.

3.2.2.3 Testing

Test results are calculated using a set of 500 images that were generated and labeled as described in Section 3.2.2.1. The backgrounds used in these images are totally different from the backgrounds used in training, and are not seen by the model prior to the testing. The model's prediction can be directly compared to the truth data. To quantify the model's performance, the distance between the predicted bounding box's center and the true bounding box's center in pixel space is measured. The CNN's error is measured in pixel space because stereo block matching generates disparity values in pixel space as well. Several test images will also be examined in Chapter IV.

3.3 CNN Application Procedure

To provide a speedup to the AAR pose estimation process, the CNN's bounding box must be used to accelerate the pose estimation process. Once the stereo images are captured, the left image is down-sampled from the original resolution to 512×386 and passed as input to the CNN. While it would be possible to perform bounding on both images, the disparities at $30m$ are only a few pixels. This means that the error from assuming both bounding boxes are the same is small enough that it does not appreciably decrease the CNN's performance. Additionally, this means the network only runs once, saving valuable computation time. The bounding box is then used to mask a pre-computed rectification map (Kaehler and Bradski give an in-depth explanation for the rectification process [9]). The captured images are remapped using this now-cropped rectification map into a final pair of rectified, undistorted, and cropped images. These images are then passed into OpenCV's stereo block matcher to generate a disparity map. Finally, the disparity map is reprojected into 3D space for use as a point cloud for pose registration. To compare the previous

pipeline with the new one, data is collected on the precision of the pose estimation process for a simulated approach and also time the point-cloud generation process for stereo camera pairs at a variety of resolutions. Appendix A gives an in-depth guide, including code, for this process.

IV. Results and Analysis

This chapter discusses the results of the ground test and examines the benefits of adding the convolutional neural network (CNN) augmentation to the stereo vision pipeline. Section 2.8 motivated this research by previewing the effects of higher resolution cameras in a virtual approach. The ground experiment supports those calculations using electro-optical (EO) camera pairs at two resolutions, as well as an infrared (IR) camera pair at a different resolution.

Section 4.1 shows the results from the ground experiment, which confirms that high-resolution imagery provides the necessary increase in pose estimation precision for automated aerial refueling (AAR). Next, Section 4.2 quantifies the deep learning model's performance. Finally, Section 4.3 examines the speedup gained from using the augmented vision pipeline.

4.1 Ground Experiment Results

Figure 16 shows the results for one approach using the IR, low-resolution EO, and high-resolution EO cameras, respectively. In this approach, the pseudo-receiver was pushed towards the pseudo-tanker as directly as possible. Note that the IR and low-resolution EO cameras struggle to find a meaningful registration at a range of $20m$, with residual errors near $0.5m$. In contrast, the high-resolution cameras have errors smaller than $0.1m$ at ranges near $35m$.

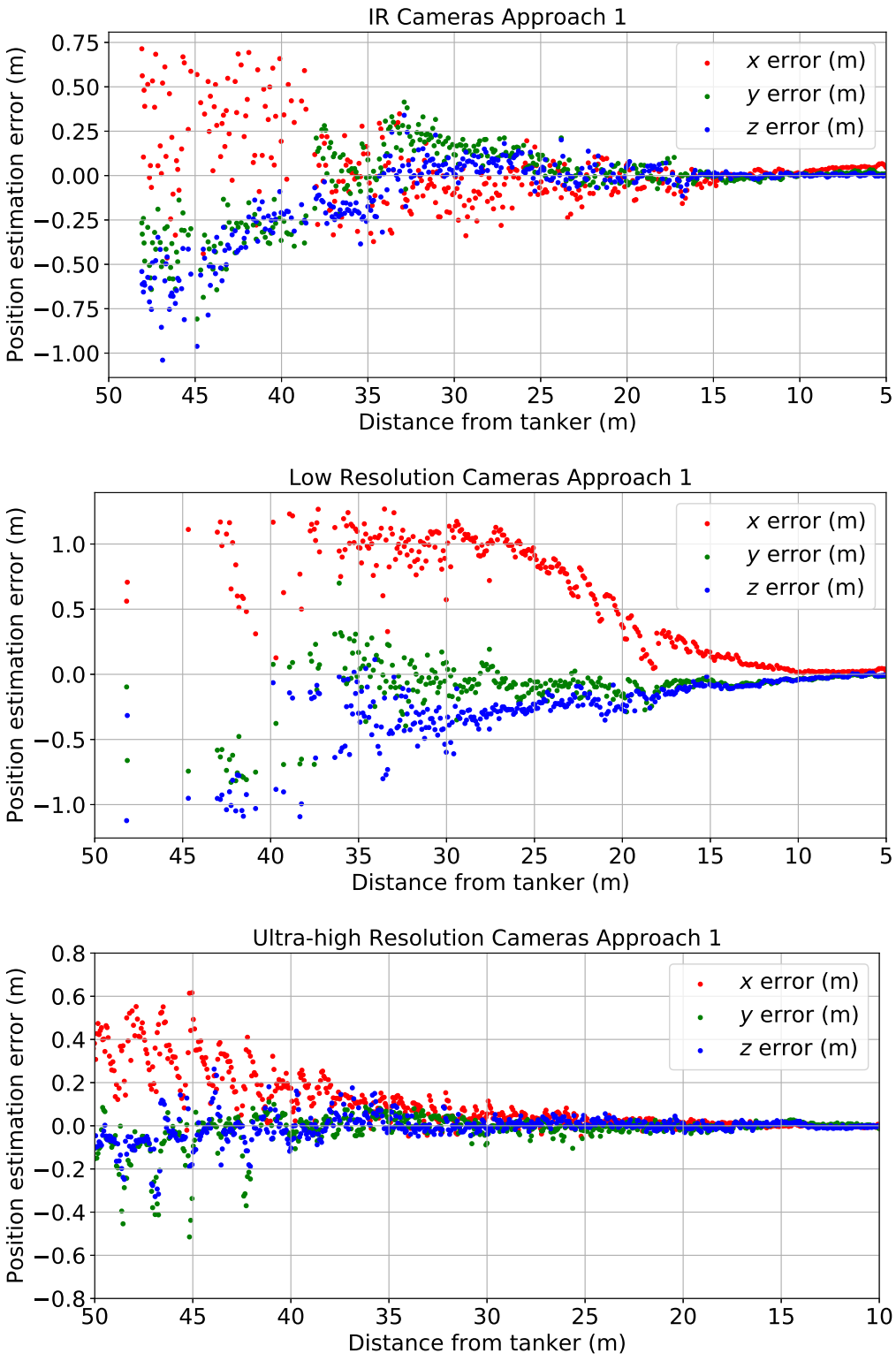


Figure 16: Residual errors for the first approach

Figure 17 shows the results for the second approach. In this approach, the pseudo-receiver was pushed a short distance and then halted for a few seconds. Since a real AAR approach might not be fully continuous, it is important to show that the technique can accurately track changes in motion as well. The results are nearly indistinguishable from the first approach, which further validates that the high-resolution cameras provide a solution accurate enough for this application.

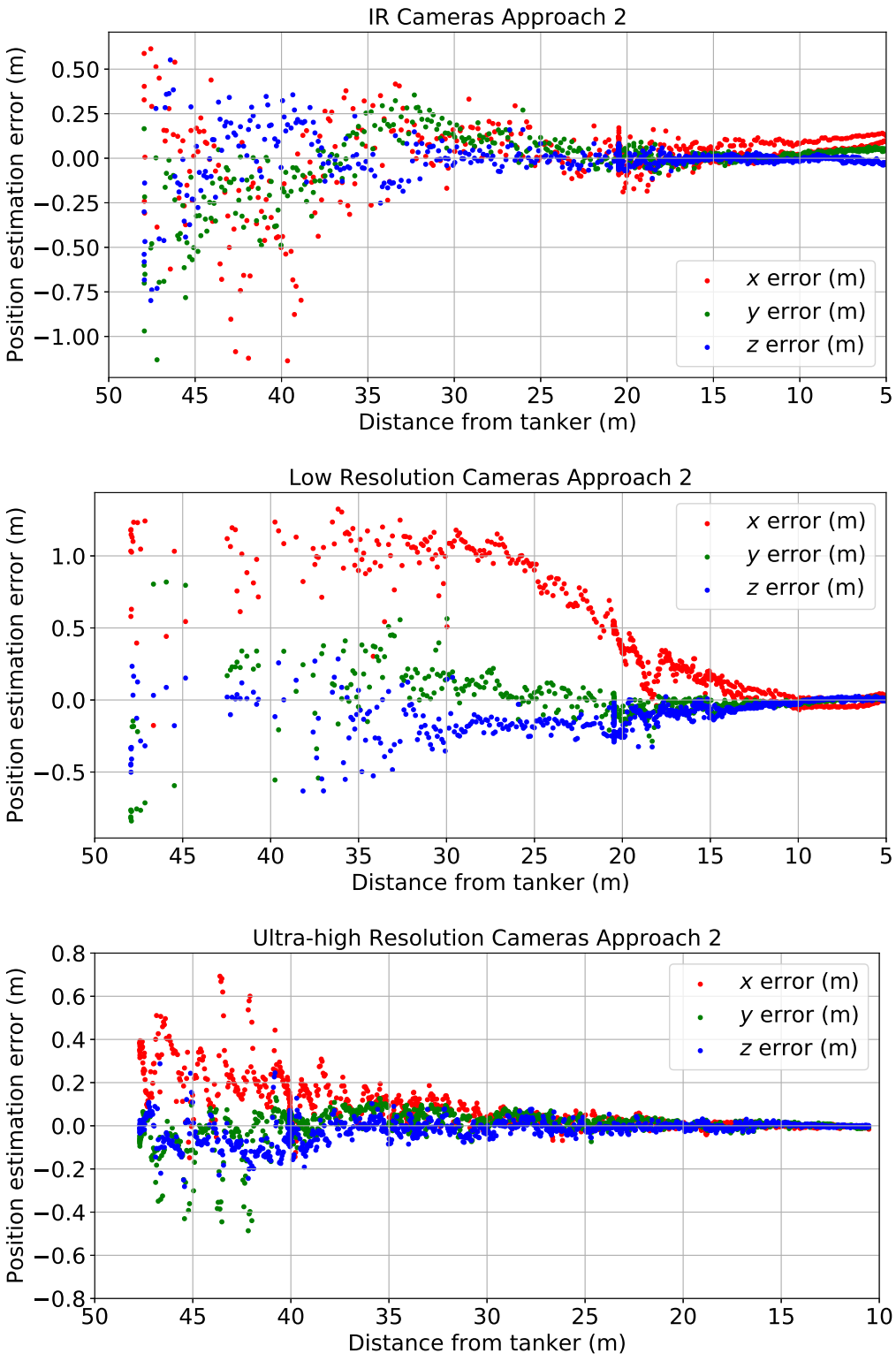


Figure 17: Residual errors for the second approach

In the approaches shown by Figure 18[1], the pseudo-receiver was moved side-to-side as it approached. This was designed to imitate an approach with suboptimal conditions that required frequent correction. Each camera system performed slightly worse; however, the high-resolution system still maintained errors smaller than $0.1m$ at the target contact point of $30m$.

[1]The receiver briefly left the IR cameras' field of view twice in this experimental approach; this is why there is no data from $35m$ to $30m$ and an uptick in error at $12m$.

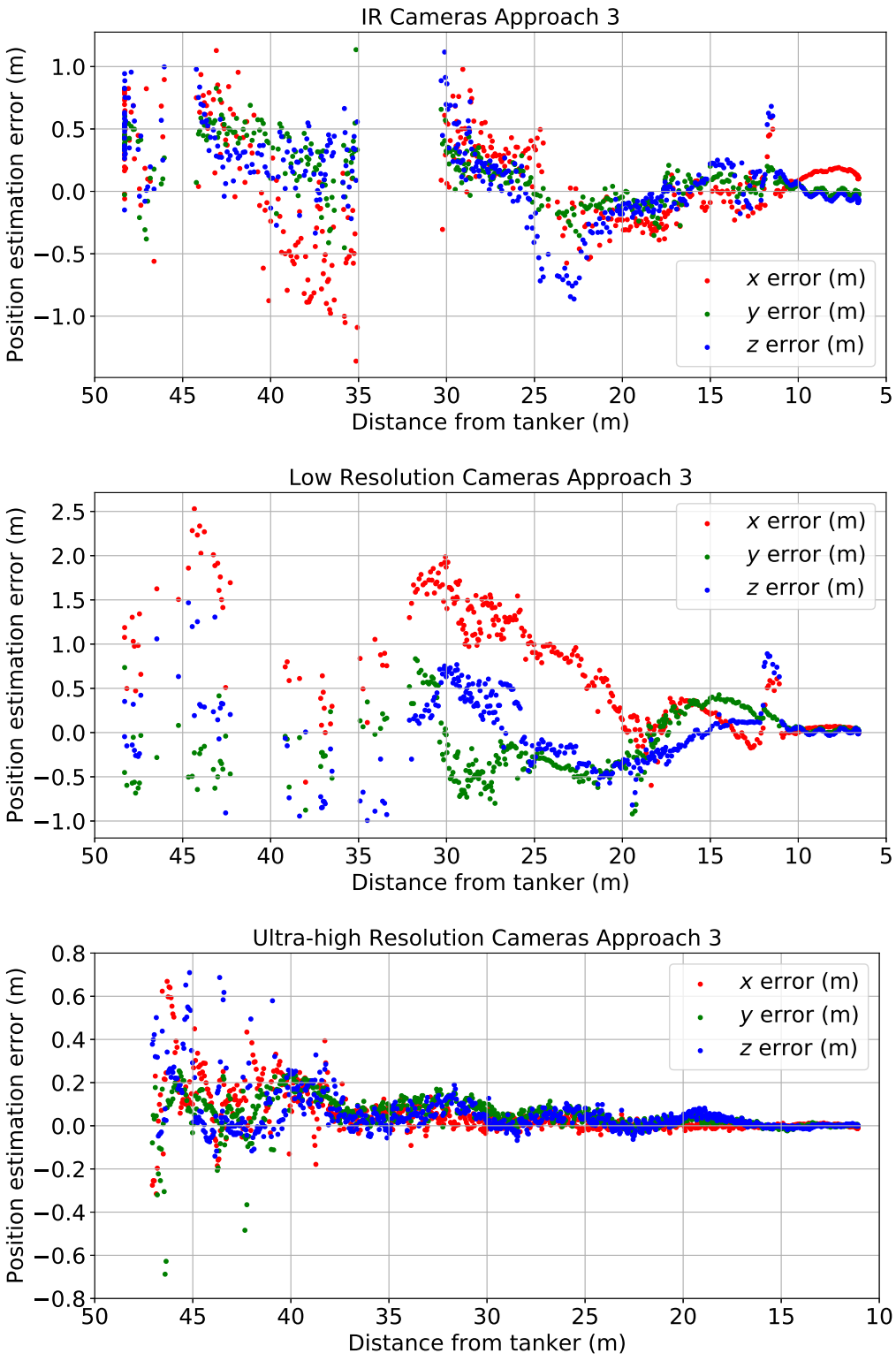


Figure 18: Residual errors for the third approach

In all, it is clear that increasing resolution does improve pose estimation accuracy. Moreover, while the low-resolution camera system struggles to obtain meaningful registrations at $20m$, the high-resolution system can perform well at ranges near $50m$. Figure 19 shows the aggregate path estimation errors for each approach. The x, y, and z components display the mean absolute error at a given range across all three approaches. The 3D error is obtained by taking the Euclidean distance between the sensed position and the truth position at each range. The error bars show a one standard deviation certainty associated with each mean. Importantly, the 3D error for the high-resolution cameras plus the error bound is less than the $10cm$ benchmark required for AAR.

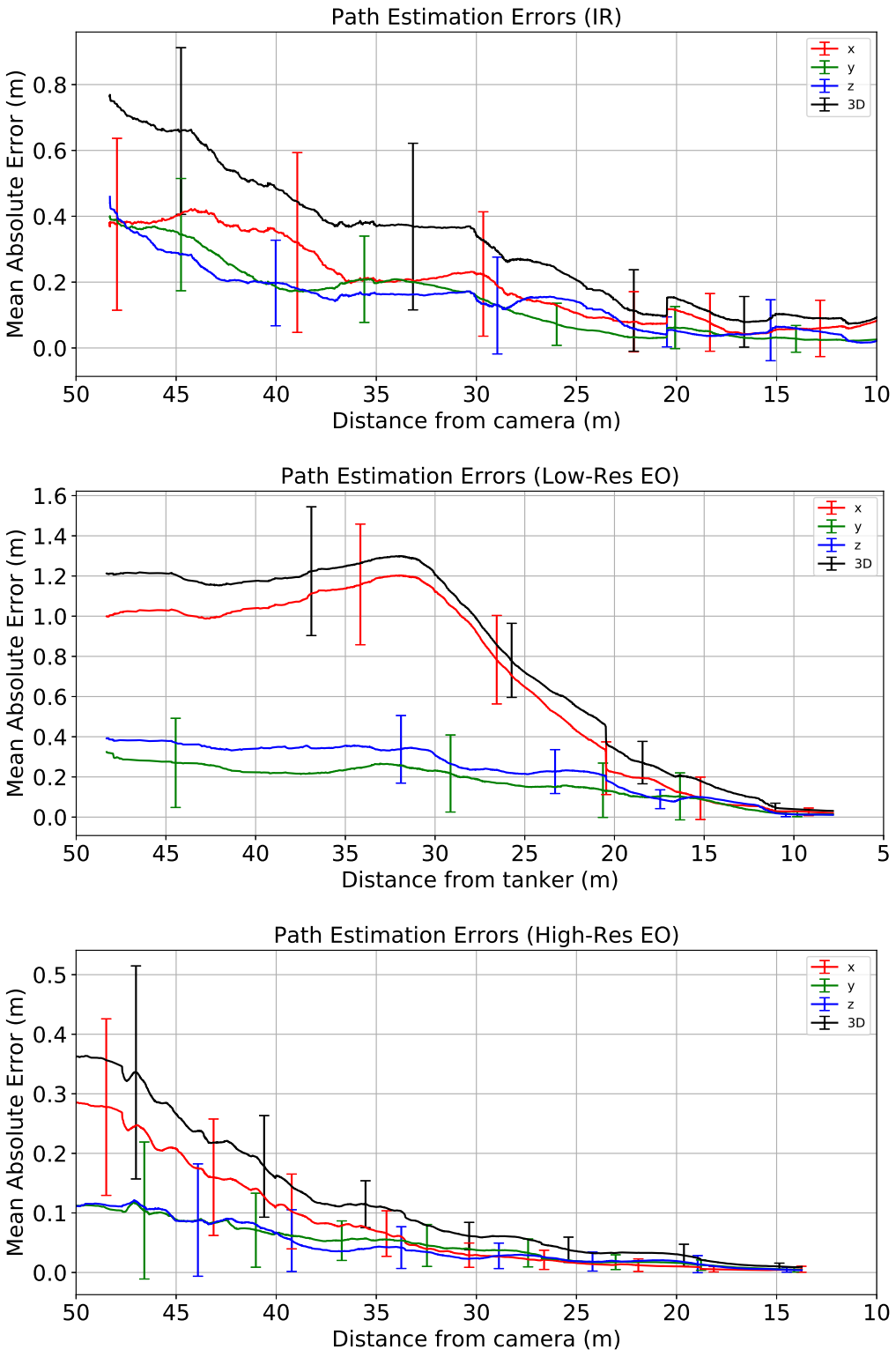


Figure 19: Aggregate errors for each camera system across all three approaches. The 3D error is the Euclidean distance from the truth position to the sensed position.

Figure 20 shows the 3D path estimation error for each camera system as a function of distance from the cameras. This clearly demonstrates the superior performance of high-resolution cameras for AAR systems. Figure 21 shows the relative error, calculated by taking the magnitude of the error vector and dividing it by the distance to the cameras. This shows that the high-resolution camera system maintains error of less than 1% for the range of interest, while the low-resolution and IR camera systems appear to level off around 3% error and 2.5% error, respectively.

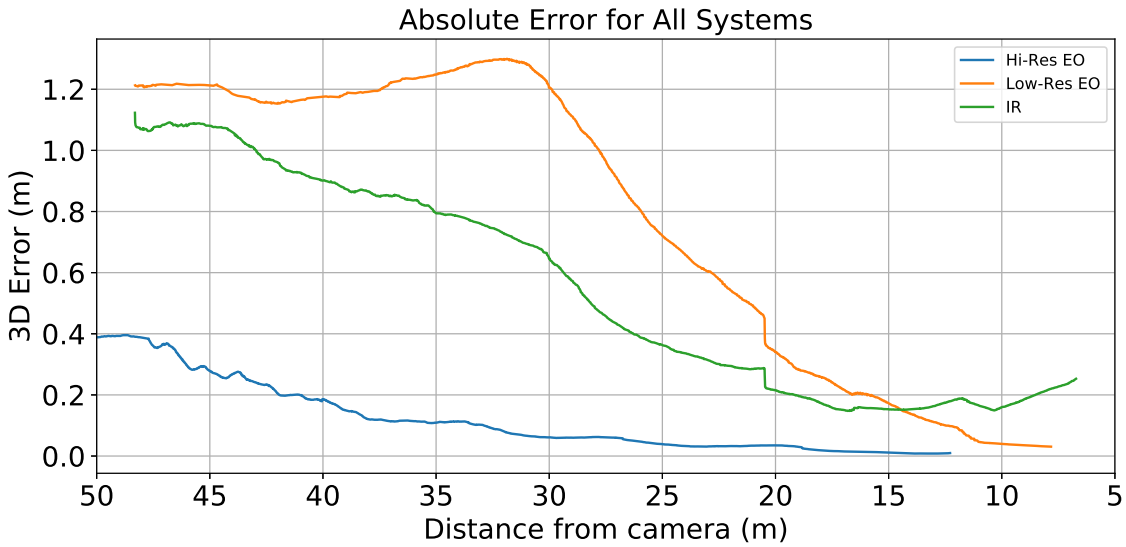


Figure 20: 3D position estimation errors for each camera system as a function of distance.

Figure 22 examines the standard deviation of the errors directly. The deviation of the errors gives important insight into the behaviors of different camera systems. While a suboptimal calibration shows residual errors for the low-resolution EO system are much higher than the IR system, examining the deviation in the errors gives the insight that the best-case scenario for those systems is actually quite similar. On the other hand, the deviation of the high-resolution EO system is much lower and much smoother as range increases.

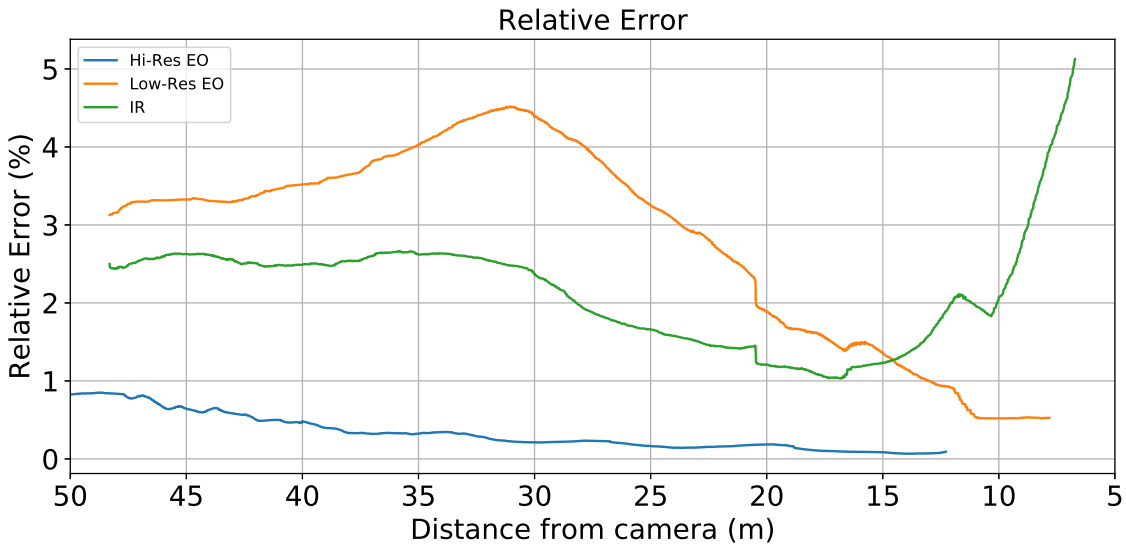


Figure 21: Relative error in each camera system as a function of distance.

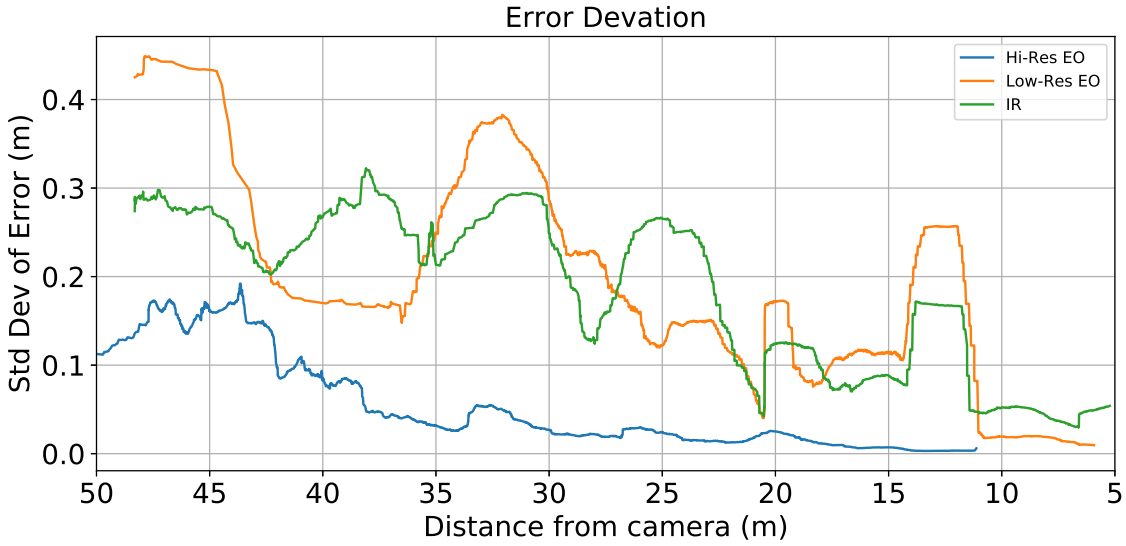


Figure 22: Standard deviation of the error in each camera system as a function of distance. The standard deviation allows us to remove lever-arm or calibration errors to compare camera systems' performance to each other in best-case scenarios.

These results indicate that our high-resolution camera system is capable of safely providing a sensed point cloud as a basis for pose estimation in AAR. However, as discussed in Chapter I, the increase in pixel count drastically increases the computation

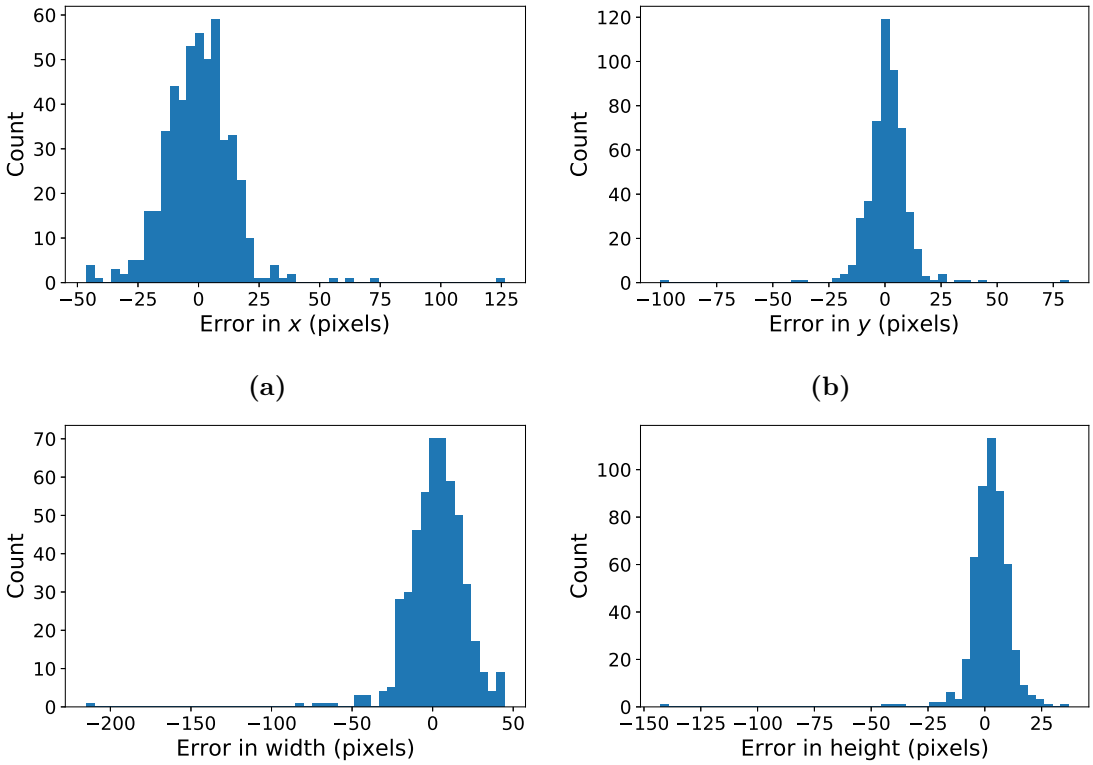
time required to generate a point cloud.

4.2 CNN Results

Table 2 shows the quantitative measures of the network's performance. root mean squared error (RMSE) and mean absolute error (MAE) are evaluated by comparing the predicted bounding box to the truth bounding box in pixel space. A few outliers slightly skew the errors; however, most are very near zero. Figure 23 shows the distribution of errors in x, y, width, and height. From these histograms, it is clear that the errors are dominated by a few outliers, and on average the network performs very well. Figure 24 shows how the errors stay fairly consistent across the entire range of the test data. Figure 25 shows the amount of the true bounding box that the predicted bounding box covers as a function of distance between the camera and the receiver. In the entire test set of 500 images, there is only one image where the predicted bounding box does not overlap the truth bounding box (a failure rate of 0.2%). Moreover, on average the model's prediction overlaps 90% of the true bounding box. These results demonstrate stable behavior throughout the refueling approach.

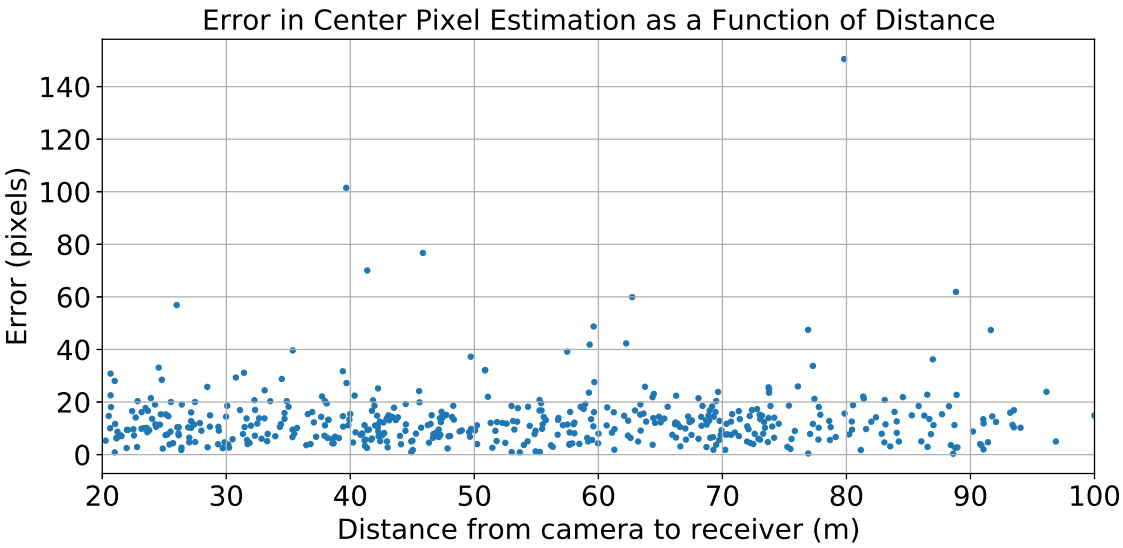
Table 2: Errors for the deep learning model (in pixels, images at 1280×960) on the test set

	x	y	$width$	$height$
RMSE	14.72	9.99	19.41	10.43
MAE	10.46	6.21	13.18	6.31

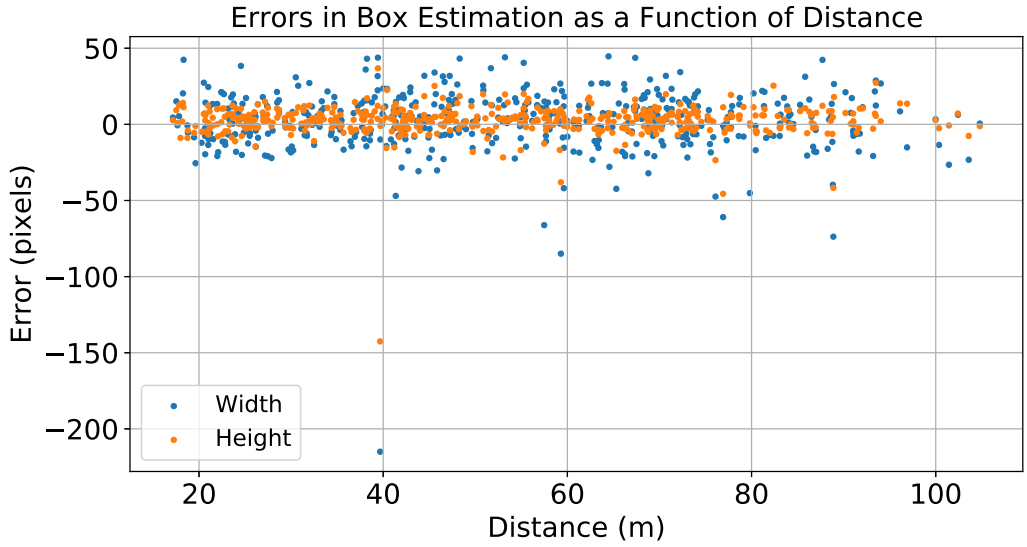


(a) **(b)** **(c)** **(d)**

Figure 23: CNN predicted bounding box errors in x, y, w, h (in pixels)



(a)



(b)

Figure 24: CNN predicted bounding box residual errors in Euclidean distance from truth box center to predicted box center and error in bounding box width and height (in pixels) as a function of distance from tanker to receiver (in meters).

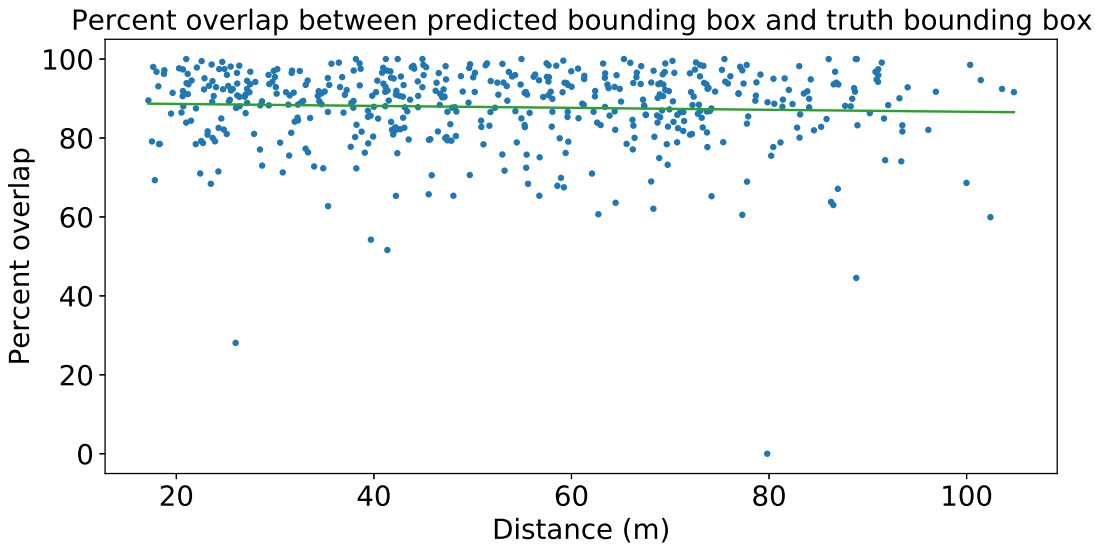
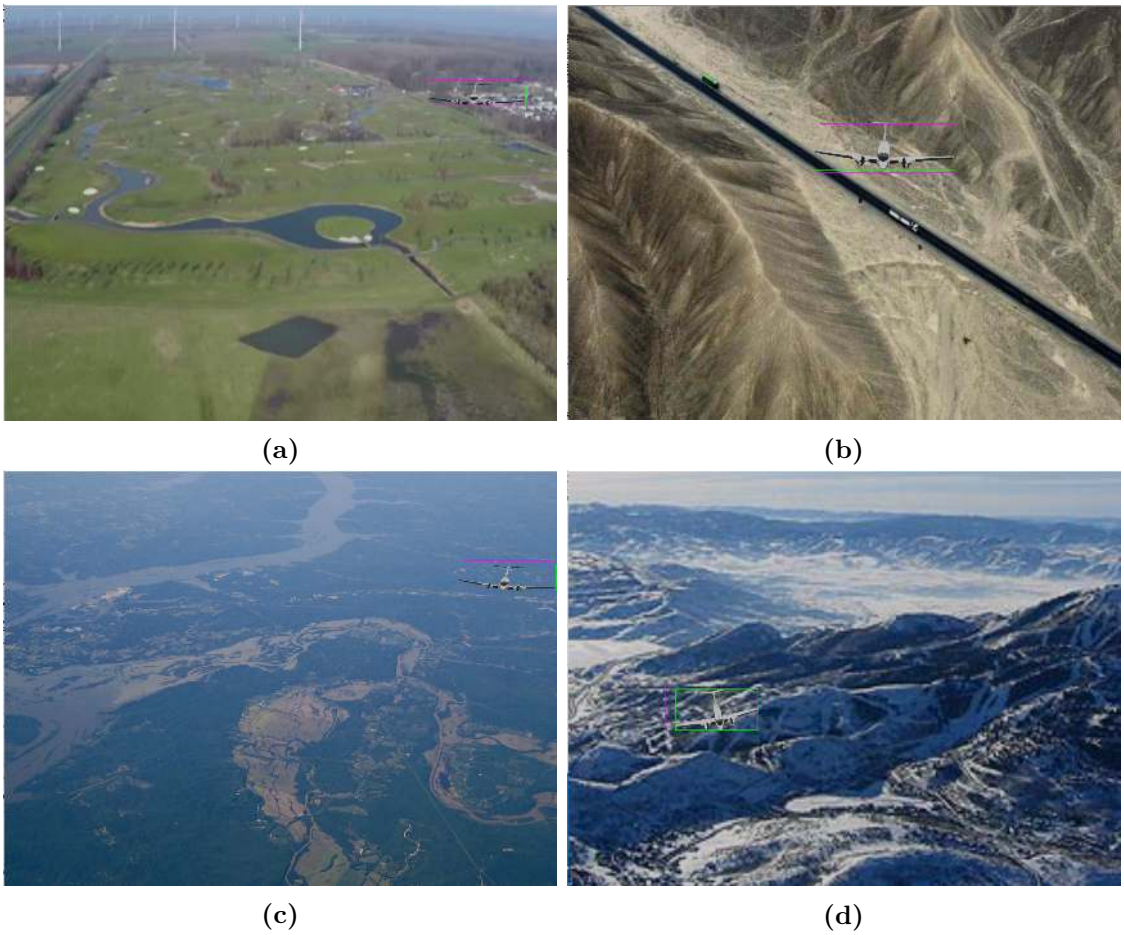


Figure 25: This measures the percent of the true bounding box that the predicted bounding box covers and shows a best fit line for convenience

Figure 26 shows four examples of the CNN's typical performance. These are consistent with the model's performance across the test data set. The model can also be used without transfer learning or other modification to evaluate images from a real flight test. Figure 27 shows an example of the model directly being used on imagery from a physical camera.



(a) **(b)**

(c) **(d)**

Figure 26: Examples of CNN model performance. The green box represents the ground-truth bounding box. The purple represents the network's predicted bounding box.

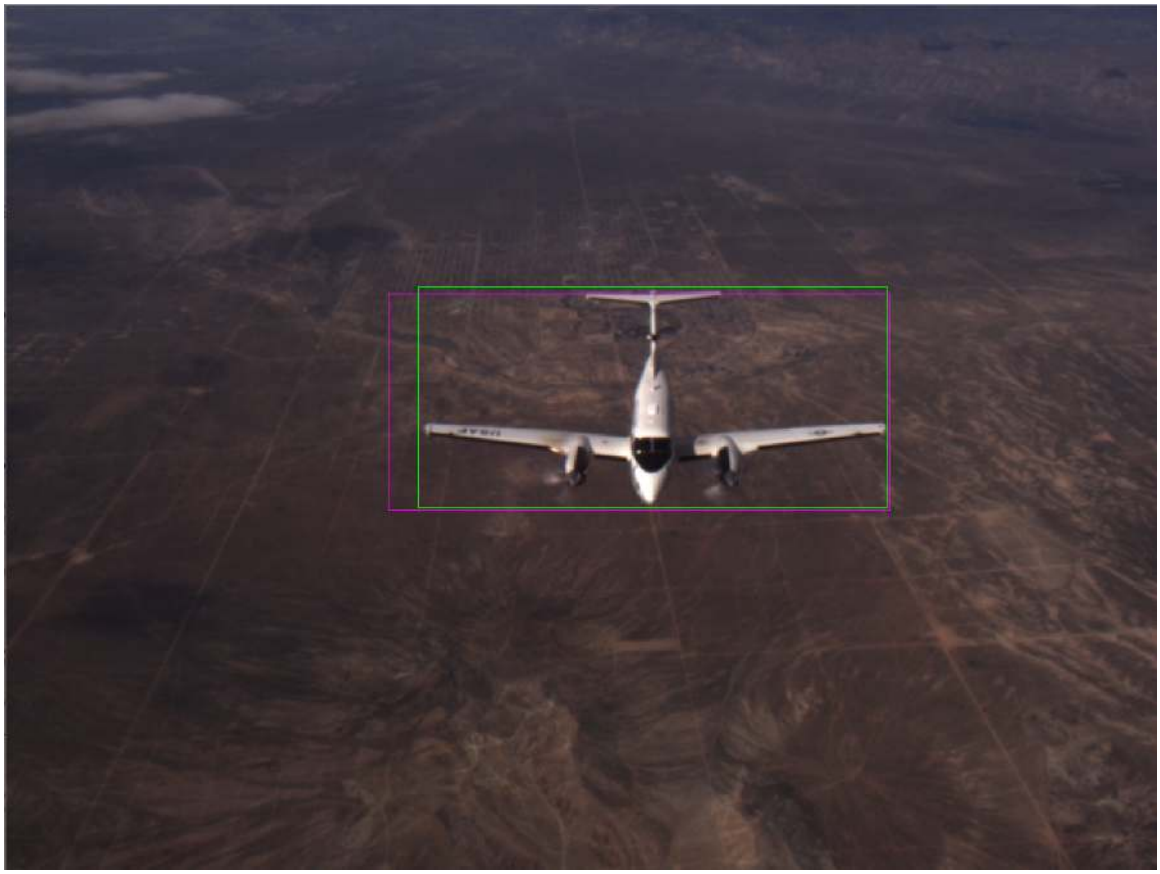


Figure 27: Performance example for the CNN model with real flight test imagery.

4.3 Point Cloud Generation Timing and Pose Estimation Precision

This subsection evaluates the CNN's performance at improving the pose estimation process for AAR. First, the affects the CNN has on the time needed to generate a 3D point cloud are analyzed. Next, the affects of using the CNN on the precision and deviation in pose estimation accuracy are examined. Table 3 shows the time required to generate a 3D point cloud at four sample resolution with and without the CNN augmentation. It is clear that the CNN provides a substantial speed-up at all resolutions. Creating a 3D point cloud for the 1280×960 image pairs is $3.6\times$ faster with the CNN augmentation. When the resolution increases to 4896×3264, the CNN gives an $11.2\times$ speedup.

Table 3: Point-cloud generation time with and without using the CNN

Resolution	**CNN Off**	**CNN On**
1280×960	36.13ms	10.57ms
1920×1440	76.77ms	14.67ms
3840×2880	357.49ms	35.03ms
4896×3264	524.28ms	46.52ms

Finally, it is important to verify that the CNN augmentation does not adversely affect pose estimation. As with the ground experiment, the path estimation error and the standard deviation of the error over the distance of the approach are shown. Figure 28 shows the 3D path estimation errors for the 1280×960 camera system and the 4896×3264 camera system with and without CNN augmentation to the vision pipeline.

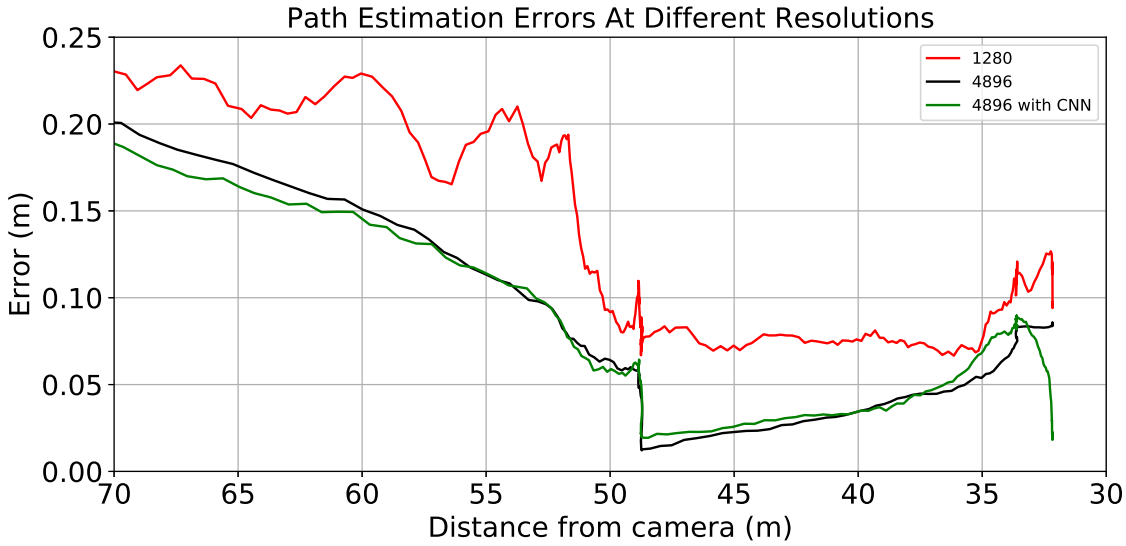


Figure 28: Path estimation error (Euclidean distance from truth position to sensed position in meters). Note that the low-resolution system's solution, even in the simulation, is consistently worse than the high-resolution system. Also note there is not an appreciable difference between the high-resolution systems error with or without the CNN augmentation. This shows the system can reliably perform as well as the original pipeline, while also gaining a large speedup.

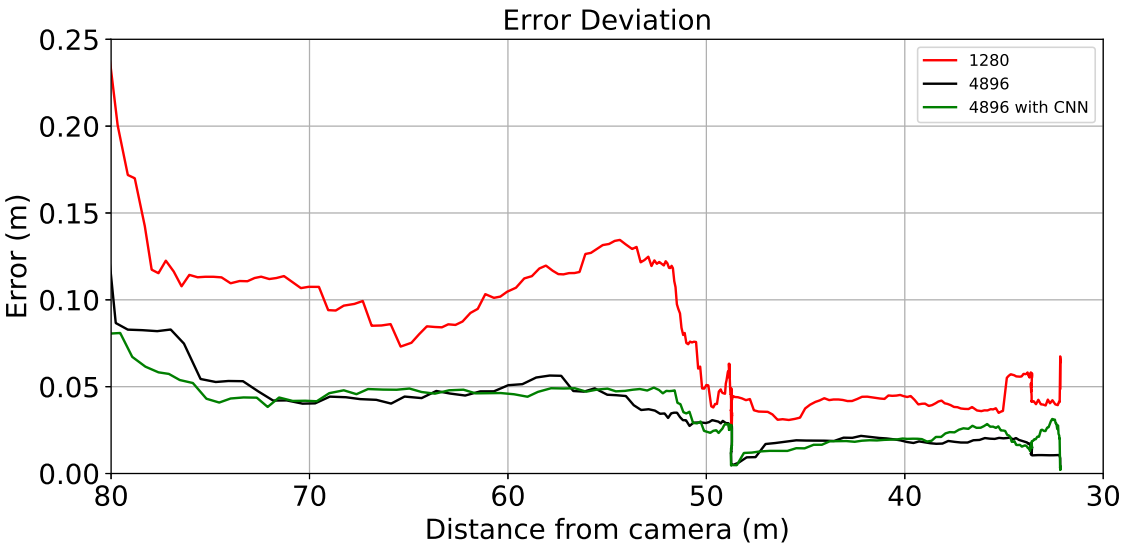


Figure 29: Standard deviation of the error of each camera system's pose estimation as a function of distance. There is no appreciable difference between the high-resolution system with and without the CNN augmentation, further validating its viability.

V. Conclusions

The results in Section 4.1 validate AFIT's stereo vision pipeline [5] to perform pose estimation for automated aerial refueling (AAR) using a high-resolution stereo camera system. The system consistently achieves 3D pose estimation errors of less than $6cm$. Based on these results, a stereo camera system with adequate resolution can safely control a receiver in the refueling envelope to make and maintain contact. However, high-resolution imagery comes with a computation-time cost.

Next, this thesis outlines a new computer vision pipeline that combines conventional stereo vision with deep learning greatly accelerate the process of generating a 3D point cloud of the receiver. It further verifies that the speedup does not decrease the precision gained from using high resolution stereo imagery. While this system was developed specifically for AAR, any real-time computer vision application could benefit from its use. For example, a convolutional neural network (CNN) can identify and label several objects of interest in a stereo image pair and then perform the pose estimation process quickly on each of them. Since the point clouds are generated from finely cropped images, the resulting 3D point clouds are already semantically segmented. This technique could be used provide benefits for many computer vision applications, including autonomous vehicles, robot navigation, or structure from motion. All of these fields could provide important capabilities to the Air Force.

The greatest remaining limitation for pose estimation speed is point cloud registration. Using a parallel iterative closest point (ICP) or development of a faster point cloud registration algorithm will be important for any future efforts to increase pose estimation rates. Further study could be done to determine if pixel-wise image segmentation could yield even greater speedups for the AAR domain. A pixel-wise mask could be used to crop the images similarly to the rectangle this thesis uses. Additionally, error in pose estimation and image segmentation could likely be greatly

reduced by leveraging the time-series nature of most relative navigation tasks. Since the receiver moves less than $2m/s$, it is generally safe to assume that its pose does not change dramatically from one frame to another. For segmentation, leveraging a recurrent neural network (RNN) to perform image segmentation may allow better tracking of objects. Future work to improve pose estimation may employ a Kalman filter.

In conclusion, AAR imposes strict requirements on a pose estimation system. High-resolution camera systems meet these requirements at a high computational cost. However, by leveraging deep learning techniques, it is possible to vastly reduce this cost. A computer vision pipeline using the techniques outlined here can provide high-precision pose estimates at long ranges. The system described here can provide the high-precision relative navigation solution for a complete AAR system.

Appendix A. Model Deployment and Usage

This appendix demonstrates how to perform real-time image segmentation in C++. Appendix 1.1 demonstrates how to deploy a pre-trained TensorFlow model in a C++ environment using the C application programming interface (API). Next, Appendix 1.2 shows how the image processing pipeline can be implemented to perform real-time cropping and generate a 3D point cloud from a stereo image pair using a CNN. Appendix 1.3 gives relevant information for future students regarding the location of different data sets and code bases.

1.1 Deploying A CNN in C++

First, the model must be trained in using either TensorFlow or Keras with a TensorFlow back end. This Python 3 function can be used to save the model into a version deployable in C and C++ environments. The output is a "frozen" model that can be used for real-time inference.

Listing A.1: Function to save a trained CNN

```
def my_freeze_graph(output_node_names, destination, name="frozen_model.pb"):
    """
    Freeze the current graph alongside its weights (converted to constants) into a protobuf file.
    :param output_node_names: The name of the output node names we are interested in
    :param destination: Destination folder or remote service (eg. gs://)
    :param name: Filename of the saved graph
    :return:

    assume "import tensorflow as tf
            import keras as K"
    """
    tf.keras.backend.set_learning_phase(0)  # set inference phase

    sess = K.get_session()
    input_graph_def = sess.graph.as_graph_def()      # get graph def proto from keras session's graph

    with sess.as_default():
        # Convert variables into constants so they will be stored into the graph def
        output_graph_def = tf.graph_util.convert_variables_to_constants(
            sess,
            input_graph_def,
            output_node_names=output_node_names)
```

```
23
24          tf.train.write_graph(graph_or_graph_def=output_graph_def, logdir=destination, name=name, as_text=False)
25
26      K.clear_session()
```

Next, the model must be read into the C++ environment. TensorFlow's C API must already be installed for this functionality. This code was compiled and tested using the C++17 standard, but should work with any C++11 or later version. The `AftrAI` class serves as a basic, generic wrapper for a TensorFlow model. The header includes several utilities the class needs to function.

Listing A.2: AftrAI.h

```
1  # pragma once
2
3  /*
4  Andrew Lee
5  5 September 2019
6  AftrAI.h
7
8  This module serves as a convenient wrapper for the TensorFlow C API,
9  allowing us to deploy ML models without knowing exactly how the API/TensorFlow work.
10 */
11
12 #include <tensorflow/c/c_api.h>
13 #include <memory>
14 #include <iostream>
15 #include <algorithm>
16 #include <cstddef>
17 #include <iterator>
18 #include <vector>
19 #include <assert.h>
20 #include <string.h>
21 #include <fstream>
22 #include <stdint.h>
23
24 namespace AI
25 {
26
27 static TF_Buffer *read_tf_buffer_from_file(const char* file);
28
29 /**
30 * A Wrapper for the C API status object.
31 */
32 class CStatus {
33 public:
34 TF_Status *ptr;
```

```
35  CStatus()
36  {
37  ptr = TF_NewStatus();
38  }
39
40  /**
41  * Dump the current error message.
42  */
43  void dump_error()const
44  {
45  std::cerr << "TF status error: " << TF_Message(ptr) << std::endl;
46  }
47
48  /**
49  * Return a boolean indicating whether there was a failure condition.
50  * @return
51  */
52  inline bool failure()const
53  {
54  return TF_GetCode(ptr) != TF_OK;
55  }
56
57  ~CStatus()
58  {
59  if (ptr)TF_DeleteStatus(ptr);
60  }
61  };
62
63  namespace detail {
64  template<class T>
65  class TFObjDeallocator;
66
67  template<>
68  struct TFObjDeallocator<TF_Status> { static void run(TF_Status *obj) { TF_DeleteStatus(obj); } };
69
70  template<>
71  struct TFObjDeallocator<TF_Graph> { static void run(TF_Graph *obj) { TF_DeleteGraph(obj); } };
72
73  template<>
74  struct TFObjDeallocator<TF_Tensor> { static void run(TF_Tensor *obj) { TF_DeleteTensor(obj); } };
75
76  template<>
77  struct TFObjDeallocator<TF_SessionOptions> { static void run(TF_SessionOptions *obj) { TF_DeleteSessionOptions(obj); }
       ↪ };
78
79  template<>
80  struct TFObjDeallocator<TF_Buffer> { static void run(TF_Buffer *obj) { TF_DeleteBuffer(obj); } };
81
82  template<>
83  struct TFObjDeallocator<TF_ImportGraphDefOptions> {
84  static void run(TF_ImportGraphDefOptions *obj) { TF_DeleteImportGraphDefOptions(obj); }
85  };
86
87  template<>
```

```
88  struct TFObjDeallocator<TF_Session> {
89  static void run(TF_Session *obj) {
90  CStatus status;
91  TF_DeleteSession(obj, status.ptr);
92  if (status.failure()) {
93  status.dump_error();
94  }
95  }
96  };
97  }
98
99  template<class T> struct TFObjDeleter {
100 void operator()(T* ptr) const {
101 detail::TFObjDeallocator<T>::run(ptr);
102 }
103 };
104
105 template<class T> struct TFObjMeta {
106 typedef std::unique_ptr<T, TFObjDeleter<T>> UniquePtr;
107 };
108
109 template<class T> typename TFObjMeta<T>::UniquePtr tf_obj_unique_ptr(T *obj) {
110 typename TFObjMeta<T>::UniquePtr ptr(obj);
111 return ptr;
112 }
113
114 class MySession {
115 public:
116 typename TFObjMeta<TF_Graph>::UniquePtr graph;
117 typename TFObjMeta<TF_Session>::UniquePtr session;
118
119 TF_Output inputs, outputs;
120 };
121
122 /**
123 * Load a GraphDef from a provided file.
124 * @param filename: The file containing the protobuf encoded GraphDef
125 * @param input_name: The name of the input placeholder
126 * @param output_name: The name of the output tensor
127 * @return
128 */
129
130 MySession* my_model_load(const char *filename, const char *input_name, const char *output_name);
131
132 template<class T> static void free_cpp_array(void* data, size_t length) {
133 delete[]((T *)data);
134 }
135
136 /**
137 * Deallocator for TF_NewTensor data.
138 * @tparam T
139 * @param data
140 * @param length
141 * @param arg
```

```
142  */
143  // Use this function if the data for the TensorFlow model is manually allocated on the heap
144  template<typename T> static void cpp_array_deallocator(void* data, size_t length, void* arg) {
145  delete[]((T *)data);
146  }
147
148  // Use this function if the data for the TensorFlow model is on the stack or stored by a smart pointer
149  static void null_deallocator(void* data, size_t length, void*arg)
150  {
151  ; // do nothing
152  }
153
154  static TF_Buffer* read_tf_buffer_from_file(const char* file);
155
156  constexpr int MY_TENSOR_SHAPE_MAX_DIM = 16;
157  struct TensorShape {
158  int64_t values[MY_TENSOR_SHAPE_MAX_DIM];
159  int dim;
160
161  int64_t size()const {
162  assert(dim >= 0);
163  int64_t v = 1;
164  for (int i = 0; i < dim; i++)v *= values[i];
165  return v;
166  }
167  };
168
169  class AftrAI
170  {
171  public:
172      enum class INPUT_CLASS { LIST, IMAGE_2D, POINTCLOUD_3D };
173
174      static std::unique_ptr<AftrAI> New(INPUT_CLASS ModelInput, TF_DataType InputDataType, size_t OutputLength);
175
176      void set_input_shape(std::array<int, 2> ListInputDimensions);
177      void set_input_shape(std::array<int, 3> Image2dInputDimensions);
178      void set_input_shape(std::array<int, 4> Pointcloud3dInputDimensions);
179
180      virtual bool valid_session() const;
181      virtual void load_model(std::string filename, std::string input_tensor_name, std::string output_tensor_name);
182
183      template<typename T>
184      std::vector<T> run_model(void* input_data) const noexcept; // run_model owns no memory. It will not delete anything.
185
186      AftrAI(const AftrAI&) = delete;
187      AftrAI& operator=(const AftrAI&) = delete;
188      virtual ~AftrAI() = default;
189  protected:
190      AftrAI() = default;
191      INPUT_CLASS ModelInputType;
192      TensorShape input_shape;
193      size_t DataSizeInBytes = 1;
194      size_t OutputLength = 0;
195      TF_DataType InputDataType = TF_DataType::TF_FLOAT;
```

```
196     std::unique_ptr<MySession> session = nullptr;
197 };
198
199 } // namespace AI
```

Listing A.3: AftrAI.cpp

```
1  // AftrAI.cpp implements the AI wrapper for TensorFlow
2
3  #include "AftrAI.h"
4  #include <array>
5  #include <vector>
6  using namespace AI;
7
8  std::unique_ptr<AftrAI> AftrAI::New(INPUT_CLASS ModelInput, TF_DataType InputDataType, size_t OutputLength)
9  {
10     auto model = std::unique_ptr<AftrAI>(new AftrAI);
11     model->ModelInputType = ModelInput;
12     model->InputDataType = InputDataType;
13     model->OutputLength = OutputLength;
14     return std::move(model);
15 }
16
17 void AftrAI::set_input_shape(std::array<int, 2> ListInputDimensions)
18 {
19     if (this->ModelInputType != INPUT_CLASS::LIST)
20     {
21         std::cout << "Input shape error. For a list/vector input, you need to specify 0: length and 1: number of
               ↪ channels (typically 1)\n"
22         << "Example expected format: {1024, 1} corresponds to a normal list with 1024 items.\n";
23         exit(1337);
24     }
25     int dtypesize = static_cast<int>(TF_DataTypeSize(this->InputDataType));
26     this->input_shape = { {1, ListInputDimensions.at(0), ListInputDimensions.at(1)}, dtypesize };
27     for (const auto& dim : ListInputDimensions)
28     {
29         this->DataSizeInBytes *= dim;
30     }
31     this->DataSizeInBytes *= dtypesize;
32 }
33
34 void AftrAI::set_input_shape(std::array<int, 3> Image2dInputDimensions)
35 {
36     if (this->ModelInputType != INPUT_CLASS::IMAGE_2D)
37     {
38         std::cout << "Input shape error. For an image input, you need to specify 0: height, 1:width, and 2: number of
               ↪ channels (typically 1)\n"
39         << "Example expected format: {960, 1280, 3} corresponds to a 3-channel (RGB/BGR/etc) image with heigh 960px and
               ↪ width 1280px.\n";
40         exit(1337);
41     }
```

```
42      int dtypesize = static_cast<int>(TF_DataTypeSize(this->InputDataType));
43      this->input_shape = { {1, Image2dInputDimensions.at(0), Image2dInputDimensions.at(1), Image2dInputDimensions.at(2)},
            ↪  dtypesize };
44      for (const auto& dim : Image2dInputDimensions)
45      {
46          this->DataSizeInBytes *= dim;
47      }
48      this->DataSizeInBytes *= dtypesize;
49  }
50
51  void AftrAI::set_input_shape(std::array<int, 4> Pointcloud3dInputDimensions)
52  {
53      if (this->ModelInputType != INPUT_CLASS::IMAGE_2D)
54      {
55          std::cout << "Input shape error. For an PC input, you need to specify 0: x shape, 1: y shape, 2: z shape, and 3:
                ↪  number of channels (typically 1)\n"
56          << "Example expected format: {10, 10, 10, 1}\n";
57          exit(1337);
58      }
59      int dtypesize = static_cast<int>(TF_DataTypeSize(this->InputDataType));
60      this->input_shape = { {1, Pointcloud3dInputDimensions.at(0), Pointcloud3dInputDimensions.at(1),
            ↪ Pointcloud3dInputDimensions.at(2), Pointcloud3dInputDimensions.at(3)}, dtypesize };
61      for (const auto& dim : Pointcloud3dInputDimensions)
62      {
63          this->DataSizeInBytes *= dim;
64      }
65      this->DataSizeInBytes *= dtypesize;
66  }
67
68  bool AftrAI::valid_session() const
69  {
70      if (!this->session)
71          return false;
72      return true;
73  }
74
75  void AftrAI::load_model(std::string filename, std::string input_tensor_name, std::string output_tensor_name)
76  {
77      this->session = std::unique_ptr<MySession>(my_model_load(filename.c_str(), input_tensor_name.c_str(),
            ↪ output_tensor_name.c_str()));
78  }
79
80  template std::vector<float> AftrAI::run_model(void* input_data) const noexcept;
81  template std::vector<double> AftrAI::run_model(void* input_data) const noexcept;
82  template std::vector<int> AftrAI::run_model(void* input_data) const noexcept;
83  template std::vector<char> AftrAI::run_model(void* input_data) const noexcept;
84  template<typename T>
85  std::vector<T> AftrAI::run_model(void* input_data) const noexcept
86  {
87      std::vector<T> ret;
88      auto input_values = tf_obj_unique_ptr(
89          TF_NewTensor(this->InputDataType, this->input_shape.values, this->input_shape.dim,
90          input_data, this->DataSizeInBytes, null_deallocator, nullptr)
91          );
```

```
92      if (!input_values)
93      {
94      std::cout << "Tensor creation failed!" << std::endl;
95      exit(17);
96      }
97      CStatus status;
98      TF_Tensor* inputs[] = { input_values.get() };
99      TF_Tensor* outputs[1] = {};
100     TF_SessionRun(this->session->session.get(), nullptr,
101     &session->inputs, inputs, 1,
102     &session->outputs, outputs, 1,
103     nullptr, 0, nullptr, status.ptr);
104     auto _output_holder = tf_obj_unique_ptr(outputs[0]);
105     if (status.failure())
106     {
107         status.dump_error();
108         exit(18);
109     }
110     TF_Tensor &output = *outputs[0];
111     if (TF_TensorType(&output) != this->InputDataType)
112     {
113         std::cout << "Error, unexpected output tensor type.\n";
114         exit(19);
115     }
116     size_t output_size = TF_TensorByteSize(&output) / TF_DataTypeSize(this->InputDataType);
117     assert(output_size == this->OutputLength);
118     auto output_array = static_cast<const T*>(TF_TensorData(&output));
119     for (int i = 0; i < output_size; i++)
120         ret.push_back(output_array[i]);
121     return ret;
122 }
123
124 static TF_Buffer* AI::read_tf_buffer_from_file(const char* file)
125 {
126     std::ifstream t(file, std::ifstream::binary);
127     t.exceptions(std::ifstream::failbit | std::ifstream::badbit);
128     t.seekg(0, std::ios::end);
129     size_t size = t.tellg();
130     auto data = std::make_unique<char[]>(size);
131     t.seekg(0);
132     t.read(data.get(), size);
133
134     TF_Buffer *buf = TF_NewBuffer();
135     buf->data = data.release();
136     buf->length = size;
137     buf->data_deallocator = free_cpp_array<char>;
138     return buf;
139 }
140
141 MySession* AI::my_model_load(const char *filename, const char *input_name, const char *output_name)
142 {
143     std::cout << "Loading model " << filename << "\n";
144     CStatus status;
145
```

```
146     auto graph = tf_obj_unique_ptr(TF_NewGraph());
147     {
148         // Load a protobuf containing a GraphDef
149         auto graph_def = tf_obj_unique_ptr(read_tf_buffer_from_file(filename));
150         if (!graph_def) {
151             return nullptr;
152         }
153 
154     auto graph_opts = tf_obj_unique_ptr(TF_NewImportGraphDefOptions());
155     TF_GraphImportGraphDef(graph.get(), graph_def.get(), graph_opts.get(), status.ptr);
156     }
157 
158     if (status.failure()) {
159         status.dump_error();
160         return nullptr;
161     }
162 
163     auto input_op = TF_GraphOperationByName(graph.get(), input_name);
164     auto output_op = TF_GraphOperationByName(graph.get(), output_name);
165     if (!input_op || !output_op)
166     {
167         return nullptr;
168     }
169 
170     auto session = std::make_unique<MySession>();
171     {
172         auto opts = tf_obj_unique_ptr(TF_NewSessionOptions());
173         session->session = tf_obj_unique_ptr(TF_NewSession(graph.get(), opts.get(), status.ptr));
174     }
175 
176     if (status.failure())
177     {
178         return nullptr;
179     }
180     assert(session);
181 
182     graph.swap(session->graph);
183     session->inputs = { input_op, 0 };
184     session->outputs = { output_op, 0 };
185 
186     return session.release();
187 }
```

The `ImageSegmentationModel` class allows further abstraction between the simulation environment and the details of the CNN's implementation. This allows a user with little knowledge of machine learning to employ the model effectively.

Listing A.4: ImageSegmentationModel.h

```cpp
#pragma once

// Andrew Lee
// 5 September 2019
// ImageSegmentationModel serves as a wrapper for TensorFlow so that the GLView does not have to interact with it directly
// based on https://github.com/aljabr0/from-keras-to-c

#include "AftrAI.h"
#include <array>
#include "opencv2/imgproc.hpp"
using namespace AI;

namespace LeeAI
{

constexpr int IMG_WIDTH = 512;
constexpr int IMG_HEIGHT = 384;
constexpr size_t OUTPUT_LENGTH = 4;
constexpr size_t NUM_CHANNELS = 1; // 1 for grayscale, 3 for RGB

class ImageSegmentationModel
{
public:
    ImageSegmentationModel();
    ImageSegmentationModel(std::string saved_model_name, std::string inputtensorname, std::string outputtensorname, int h, int w);

    void load_model_from_file(std::string filename, std::string inputtensorname, std::string outputtensorname);
    void set_native_image_dimensions(int h, int w);

    std::array<float, 4> bound_C12(const cv::Mat& img) const;
    virtual ~ImageSegmentationModel() = default;
protected:
    std::unique_ptr<AftrAI> tensorflow_model = nullptr;
    std::unique_ptr<MySession> session = nullptr;
    int NATIVE_W = 1280;
    int NATIVE_H = 960;
};


} // namespace LeeAI
```

Listing A.5: ImageSegmentationModel.cpp

```cpp
#include "ImageSegmentationModel.h"
using namespace LeeAI;

#include "opencv2/imgproc.hpp"


```

```
7  ImageSegmentationModel::ImageSegmentationModel()
8  {
9      this->tensorflow_model = AftrAI::New(AftrAI::INPUT_CLASS::IMAGE_2D, TF_DataType::TF_FLOAT, OUTPUT_LENGTH);
10     this->tensorflow_model->set_input_shape(std::array<int, 3>({ IMG_HEIGHT, IMG_WIDTH, NUM_CHANNELS }));
11 }
12
13 ImageSegmentationModel::ImageSegmentationModel(std::string saved_model_name, std::string inputtensorname, std::string
       ↪ outputtensorname, int h, int w)
14 {
15     this->tensorflow_model = AftrAI::New(AftrAI::INPUT_CLASS::IMAGE_2D, TF_DataType::TF_FLOAT, 4);
16     this->tensorflow_model->set_input_shape(std::array<int,3>({ IMG_HEIGHT, IMG_WIDTH, NUM_CHANNELS }));
17     assert(h / w == IMG_HEIGHT / IMG_WIDTH); // enforce aspect ratio
18     this->tensorflow_model->load_model(saved_model_name, inputtensorname, outputtensorname);
19     this->NATIVE_W = w;
20     this->NATIVE_H = h;
21 }
22
23 void ImageSegmentationModel::load_model_from_file(std::string filename, std::string inputtensorname, std::string
       ↪ outputtensorname)
24 {
25     this->tensorflow_model->load_model(filename, inputtensorname, outputtensorname);
26 }
27
28 void ImageSegmentationModel::set_native_image_dimensions(int h, int w)
29 {
30     assert(h / w == IMG_HEIGHT / IMG_WIDTH);
31     this->NATIVE_H = h;
32     this->NATIVE_W = w;
33 }
34
35 std::array<float, 4> ImageSegmentationModel::bound_C12(const cv::Mat& img) const
36 {
37     assert(this->tensorflow_model->valid_session());
38     std::array<float, 4> retval={ -1.0f,-1.0f,-1.0f,-1.0f };
39     cv::Mat reformat;
40     cv::resize(img, reformat, cv::Size(IMG_WIDTH, IMG_HEIGHT));
41     reformat.convertTo(reformat, CV_32FC1, 1.0/255.0);
42     auto output = this->tensorflow_model->run_model<float>(static_cast<void*>(reformat.data));
43     retval.at(0) = output.at(0);
44     retval.at(1) = output.at(1);
45     retval.at(2) = output.at(2);
46     retval.at(3) = output.at(3);
47     return retval;
48 }
```

To use the `ImageSegmentationModel`, it must be properly initialized. In the AftrBurner Engine, `ManagerEnvironmentConfiguration::getVariableValue()` allows the proper values to be loaded from a configuration file. You could also call the constructor directly, instead of making a unique pointer; this choice is generally stylistic.

Listing A.6: Initialize ImageSegmentationModel

```
AI = std::make_unique<LeeAI::ImageSegmentationModel>(ManagerEnvironmentConfiguration::getVariableValue("CNNmodelFile"),
"input_1", "out/BiasAdd",
AftrUtilities::toInt(ManagerEnvironmentConfiguration::getVariableValue("stereoFrustumVertSensorPixels")),
AftrUtilities::toInt(ManagerEnvironmentConfiguration::getVariableValue("stereoFrustumHorzSensorPixels")));
```

1.2 Image Processing Implementation

Once the image segmentation model has been loaded as demonstrated in Appendix 1.1, a function such as the one below can be used to generate a disparity map and 3D re-projection from the cropped images.

Listing A.7: Example function that performs image segmentation

```
// assume library inclusions, such as OpenCV
// map1x, map2x, map1y, map2y are "cv::Mat"s pre-computed to rectify images; assumed global here
// AI is a properly instantiated ImageSegmentationModel
// horzSensorPixels and vertSensorPixels are known values for the cameras
cv::Mat CropStereoImagesAndProject3D(leftRawImg, rightRawImg)
{
    boundingBox = AI->bound_C12(leftRawImg);
    auto boundingBoxAsRectangle = cv::Rect(cv::Point((int)(boundingBox.at(0)*horzSensorPixels - boundingBox.at(2)*
        ↪ horzSensorPixels / 2), (int)(boundingBox.at(1)*vertSensorPixels - boundingBox.at(3)*vertSensorPixels / 2)),
    cv::Point((int)(boundingBox.at(0)*horzSensorPixels + boundingBox.at(2)*horzSensorPixels / 2), (int)(boundingBox.at
        ↪ (1)*vertSensorPixels + boundingBox.at(3)*vertSensorPixels / 2)));

    cv::Mat leftStereoImg;
    cv::Mat rightStereoImg;
    // Crop the rectification maps using coordinates from raw images
    auto cropped_map1x = map1x(boundingBoxAsRectangle);
    auto cropped_map1y = map1y(boundingBoxAsRectangle);
    auto cropped_map2x = map2x(boundingBoxAsRectangle);
    auto cropped_map2y = map2y(boundingBoxAsRectangle);

    cv::remap(leftRawImg, leftStereoImg, cropped_map1x, cropped_map1y, cv::INTER_LINEAR);
    cv::remap(rightRawImg, rightStereoImg, cropped_map2x, cropped_map2y, cv::INTER_LINEAR);

    // perform stereo block matching
    cv::Mat bmDisparity = cv::Mat(leftStereoImg.rows, leftStereoImg.cols, CV_16S);
    // numDisparities and SADWindowSize are constants
    cv::Ptr<cv::StereoBM> stereoBM = cv::StereoBM::create(numDisparities, SADWindowSize);
    stereoBM->compute(leftStereoImg, rightStereoImg, bmDisparity); //Output disparity values are implicitly multiplied
        ↪ by 16.
    // Q is a reprojection DCM
    cv::Mat reprojection;
    cv::Mat qCoordinateTransform(4,4, CV_64F);
    qCoordinateTransform.at<double>(0, 0) = 1.0;
    qCoordinateTransform.at<double>(0, 1) = 0.0;
```

```
32      qCoordinateTransform.at<double>(0, 2) = 0.0;
33      qCoordinateTransform.at<double>(0, 3) = (boundingBox.at(0) - boundingBox.at(2)/2)*horzSensorPixels;
34      qCoordinateTransform.at<double>(1, 0) = 0.0;
35      qCoordinateTransform.at<double>(1, 1) = 1.0;
36      qCoordinateTransform.at<double>(1, 2) = 0.0;
37      qCoordinateTransform.at<double>(1, 3) = (boundingBox.at(1)-boundingBox.at(3)/2)*vertSensorPixels;
38      qCoordinateTransform.at<double>(2, 0) = 0.0;
39      qCoordinateTransform.at<double>(2, 1) = 0.0;
40      qCoordinateTransform.at<double>(2, 2) = 1.0;
41      qCoordinateTransform.at<double>(2, 3) = 0.0;
42      qCoordinateTransform.at<double>(3, 0) = 0.0;
43      qCoordinateTransform.at<double>(3, 1) = 0.0;
44      qCoordinateTransform.at<double>(3, 2) = 0.0;
45      qCoordinateTransform.at<double>(3, 3) = 1.0;
46      cv::Mat cropped_Q = Q * qCoordinateTransform;
47      cv::reprojectImageTo3D(bmDisparity, reprojection, cropped_Q, false, CV_32F)
48
49      return reprojection;
50  }
```

1.3 Repository Information

All deployed code for `AftrAI` and `ImageSegmentationModel` can be found on the Lee_MLProject branch of the AARViz repository maintained by Dr. Nykl. This includes the CMake files that link to the TensorFlow library. Instructions for installing TensorFlow are located in the AAR share drive. Currently, that is located at /aar_folder/stud/AndrewLee/TensorFlow Install Guide/.

Data processing was generally performed in Python 3. Initial experimentation with calibration is located in the AAR repository at "aar/students/andrew.lee/Lever-arm Bias". The data for the journal article can be generated using the AARViz `yardTrashBranch`. This code was not designed to be maintainable: **only use it if you must regenerate data for some reason**. The data and the scripts responsible for processing it into graphic form for the journal article and thesis are located in "aar/students/andrew.lee/Journal Article".

Bibliography

1. Stephen Losey. KC-46 refueling system flaws will take years to fix and cost hundreds of millions, GAO says, 2019.

2. Camera Calibration and 3D Reconstruction — OpenCV 2.4.13.7 documentation.

3. William Dallmann. *INFRARED AND ELECTRO-OPTICAL STEREO VISION FOR AUTOMATED AERIAL REFUELING*. PhD thesis, Air Force Institute of Technology, 2019.

4. Arne Nordmann. Epipolar geometry.svg, 2006.

5. Christopher Parsons, Zachary Paulson, Scott Nykl, William Dallman, Brian G. Woolley, and John Pecarina. Analysis of Simulated Imagery for Real-Time Vision-Based Automated Aerial Refueling. *J. Aerosp. Inf. Syst.*, 16(3):77–93, mar 2019.

6. Mario Luca Fravolini, Giampiero Campa, and Marcello R. Napolitano. Modelling and performance analysis of a machine vision-based semi-autonomous aerial refuelling. *Int. J. Model. Identif. Control*, 3(4):357, 2008.

7. Kevin Liu, Christopher Moore, Robert Buchler, Phil Bruner, Alex Fax, Jacob Hinchman, Ba Nguyen, David Nelson, Fred Ventrone, and Brian Thorward. Precision relative navigation solution for autonomous operations in close proximity. *Rec. - IEEE PLANS, Position Locat. Navig. Symp.*, pages 1246–1251, 2008.

8. S.C. de Vries. UAVs and control delays. Technical report, TNO Defence, Security and Safety, Stoesterberg (Netherlands), 2005.

9. Adrian Kaehler and Gary Bradski. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*. O'Reilly Media, Sebastopol, California, 1 edition, 2016.

10. Daniel Scharstein. A Taxonomy and Evaluation of Dense Two-Frame Stereo. *Int. J. Comput. Vis.*, 47(1):7–42, 2002.

11. Niall O'Mahony, Sean Campbell, Anderson Carvalho, Suman Harapanahalli, Gustavo Velasco Hernandez, Lenka Krpalkova, Daniel Riordan, and Joseph Walsh. Deep Learning vs. Traditional Computer Vision. *Adv. Intell. Syst. Comput.*, 943(Cv):128–144, 2020.

12. Alex Krizhevsky. Convolutional Deep Belief Networks on CIFAR-10. In *Adv. Neural Inf. Process. Syst.*, 2010.

13. Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely Connected Convolutional Networks. In *IEEE Conf. Comput. Vis. Pattern Recognit.*, pages 4700–4708, 2017.

14. Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. In *IEEE Conf. Comput. Vis. Pattern Recognit.*, pages 779–788, 2016.

15. Joseph Redmon and Ali Farhadi. YOLO9000: Better, Faster, Stronger. In *IEEE Conf. Comput. Vis. Pattern Recognit.*, pages 7263–7271, 2017.

16. Zhong-Qiu Zhao, Peng Zheng, Shou-Tao Xu, and Xindong Wu. Object Detection with Deep Learning: A Review. *IEEE Trans. Neural Networks Learn. Syst.*, 30(11):3212–3232, 2019.

17. Andy Lee. Comparing Deep Neural Networks and Traditional Vision Algorithms in Mobile Robotics. Technical report.

18. Brady Zhou, Philipp Krähenbühl, and Vladlen Koltun. Does Computer Vision Matter for Action? *Sci. Robot.*, 4(30), 2019.

19. A. Garcia-Garcia, F. Gomez-Donoso, J. Garcia-Rodriguez, S. Orts-Escolano, M. Cazorla, and J. Azorin-Lopez. PointNet: A 3D Convolutional Neural Network for real-time object class recognition. In *Proc. Int. Jt. Conf. Neural Networks*, volume 2016-Octob, pages 1578–1584. Institute of Electrical and Electronics Engineers Inc., oct 2016.

20. Antonio Torralba and Alexei A. Efros. Unbiased look at dataset bias. In *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pages 1521–1528. IEEE Computer Society, 2011.

21. David Vazquez, Antonio M. Lopez, Javier Marin, Daniel Ponsa, and David Geronimo. Virtual and real world adaptationfor pedestrian detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 36(4):797–809, 2014.

22. Manik Goyal, Param Rajpura, Hristo Bojinov, and Ravi Hegde. Dataset Augmentation with Synthetic Images Improves Semantic Segmentation. In *Natl. Conf. Comput. Vision, Pattern Recognition, Image Process. Graph.*, pages 348–359, 2018.

23. German Ros, Laura Sellart, Joanna Materzynska, David Vazquez, and Antonio M Lopez. The SYNTHIA Dataset: A Large Collection of Synthetic Images for Semantic Segmentation of Urban Scenes. In *IEEE Conf. Comput. Vis. Pattern Recognit.*, pages 3234–3243, 2016.

24. Xin Zhang, Zhiguo Jiang, and Haopeng Zhang. Real-time 6D pose estimation from a single RGB image. *Image Vis. Comput.*, 89:1–11, sep 2019.

25. Pasquale Ferrara, Alessandro Piva, Fabrizio Argenti, Junya Kusuno, Marta Niccolini, Matteo Ragaglia, and Francesca Uccheddu. Wide-angle and long-range

real time pose estimation: A comparison between monocular and stereo vision systems. *J. Vis. Commun. Image Represent.*, 48:159–168, oct 2017.

26. Mario L. Fravolini, Marco Mammarella, Giampiero Campa, Marcello R. Napolitano, and Mario Perhinschi. Machine vision algorithms for autonomous aerial refueling for UAVs using the USAF refueling boom method. *Stud. Comput. Intell.*, 304:95–138, 2010.

27. G. K. L. Tam, Yonghuai Liu, F. C. Langbein, Zhi-Quan Cheng, D. Marshall, P. L. Rosin, R. R. Martin, Xian-Fang Sun, and Yu-Kun Lai. Registration of 3D Point Clouds and Meshes: A Survey from Rigid to Nonrigid. *IEEE Trans. Vis. Comput. Graph.*, 19(7):1199–1217, 2012.

28. Paul Besl and Neil McKay. Method for Registration of 3-D Shapes. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(2):239 – 256, 1992.

29. Chad Mourning, Scott Nykl, Huihui Xu, David Chelberg, and Jundong Liu. GPU acceleration of robust point matching. In *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, volume 6455 LNCS, pages 417–426, 2010.

30. Qian Yi Zhou, Jaesik Park, and Vladlen Koltun. Fast global registration. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, 9906 LNCS:766–782, 2016.

31. Gil Elbaz, Tamar Avraham, and Anath Fischer. 3D Point Cloud Registration for Localization using a Deep Neural Network Auto-Encoder. In *IEEE Conf. Comput. Vis. Pattern Recognit.*, pages 4631–4640, 2017.

32. Giampiero Campa, Marcello R. Napolitano, and Mario L. Fravolini. Simulation

environment for machine vision based aerial refueling for UAVs. *IEEE Trans. Aerosp. Electron. Syst.*, 45(1):138–151, 2009.

33. Bradley French and Scott Nykl. Determining Virtually Simulated Aerial Refueling Fidelity using Physically Collected Stereo Vision Images and DGPS-based Truth Data. In *Jt. Navig. Conf.* Institute of Navigation, 2019.

34. David Gallup, Jan-Michael Frahm, Philippos Mordohai, and Marc Pollefeys. Variable Baseline/Resolution Stereo Project AutoVision: Localization and 3D Scene Perception for an Autonomous Vehicle with a Multi-Camera System View project CNN based Autonomous Driving View project Variable Baseline/Resolution Stereo. In *IEEE Conf. Comput. Vis. Pattern Recognit.*, 2008.

35. Scott Nykl, Chad Mourning, Mitchell Leitch, David Chelberg, Teresa Franklin, and Chang Liu. An overview of the STEAMiE Educational game engine. In *Proc. - Front. Educ. Conf. FIE*, 2008.

36. Nicholas Seydel, Will Dallmann, and Scott Nykl. Visualizing Behaviors when Using Real vs Synthetic Imagery for Computer Vision. In *Int. Conf. Sci. Comput.*, 2018.

37. James D. Anderson, Scott Nykl, and Thomas Wischgoll. Augmenting Flight Imagery from Aerial Refueling. In *Int. Symp. Vis. Comput.*, pages 154–165, 2019.

Acronyms

AAR automated aerial refueling. iv, 1, 2, 3, 4, 5, 6, 7, 15, 16, 18, 19, 28, 30, 33, 35, 37, 41, 43, 44, 50, 53, 54, 1

AFIT Air Force Institute of Technology. iv, 1

API application programming interface. 22, 55

CNN convolutional neural network. 2, 3, 4, 14, 15, 21, 29, 31, 33, 35, 53, 55

DCM direction cosine matrix. 9

DGPS differential GPS. 24, 25, 26

EO electro-optical. 3, 18, 22, 23, 24, 35, 43

FGR fast global registration. 16

GPS Global Positioning System. 1

ICP iterative closest point. 16, 21, 24, 25, 28, 53

IMU inertial measurement unit. 6

IR infrared. 19, 22, 23, 24, 35, 43

MAE mean absolute error. 17, 45

MSAA multi-sample anti-aliasing. 28

RMSE root mean squared error. 45

RNN recurrent neural network. 54

RPAs remotely piloted aircraft. iv, 1, 7, 1

USAF United States Air Force. 1, 5

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
19–03–2020	Master's Thesis	Sept 2018 — Mar 2020

4. TITLE AND SUBTITLE

Object Detection with Deep Learning to
Accelerate Pose Estimation for Automated Aerial Refueling

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Andrew T. Lee

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-20-M-035

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RQQC
Dan Schreiter
WPAFB OH 45433-7765
COMM 937-938-7765
Email: dan.schreiter@us.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/RQQC

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

RPAs cannot currently refuel during flight because the latency between the pilot and the aircraft is too great to safely perform aerial refueling maneuvers. However, an AAR system removes this limitation by allowing the tanker to directly control the RPA. The tanker quickly finding the relative position and orientation (pose) of the approaching aircraft is the first step to create an AAR system. Previous work at AFIT demonstrates that stereo camera systems provide robust pose estimation capability. This thesis first extends that work by examining the effects of the cameras' resolution on the quality of pose estimation. Next, it demonstrates a deep learning approach to accelerate the pose estimation process. The results show that this pose estimation process is precise and fast enough to safely perform AAR.

15. SUBJECT TERMS

Aerial Refueling, Deep Learning, Computer Vision

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Scott L. Nykl, AFIT/ENG
U	U	U	UU	87	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x4395 scott.nykl@afit.edu