CSCI 370 Final Report

Capturers

Xavier Cotton
Jaxon Schauer
David Munro
Keenan Buckley

Revised Dec 5th, 2023

CSCI 370 Fall 2023

Ms. Chloe Ross

Table 1: Revision history

Revision	Date	Comments
New	8-31-23	Completed Sections: I. Introduction II. Functional Requirements III. Non Functional Requirements IV. Risks V. Definition of Done
Rev – 2	9-15-23	Completed Sections: VI. System Architecture
Rev – 3	10-22-23	Completed Sections: VII. Software Test and Quality
Rev – 4	11-10-23	Completed Sections: VIII. Project Ethical Considerations IX. Project Completion Status X. Future Work XI. Lessons Learned XII. Acknowledgements XIII. Team Profile
Rev – 5	12-4-23	Revised Sections: I. Introduction II. Functional Requirements III. Non Functional Requirements IV. Risks V. Definition of Done VI. System Architecture VII. Software Test and Quality VIII. Project Completion Status IX. Future Work X. Lessons Learned XI. Acknowledgements XII. Team Profile References Appendix A
Rev – 6	12-9-23	Revised Sections: I. Introduction II. Functional Requirements III. Non Functional Requirements IV. Risks V. Definition of Done VI. System Architecture VII. Software Test and Quality VIII. Project Completion Status Appendix A

Table of Contents

I. Introduction

In the rapidly advancing realm of technology, our team took on a project posed by Stratom to pave the way for the automation of vehicle refueling by developing a system that can accurately recognize fuel caps on vehicles, ensuring a seamless and efficient refueling process.

Project Scope and Goal

Our project's high-level scope involved the integration of cutting-edge technologies to enable autonomous vehicle refueling through the recognition of fuel caps. By leveraging machine learning (ML) algorithms and the capabilities of a RGBD (Red Green Blue Depth) camera, we aimed to create a system that can not only identify fuel caps but also provide real-time information to facilitate automated refueling procedures

Client and Motivation

Stratom, a prominent player in the automotive sector, is the driving force behind this endeavor. Recognizing the potential for automation to transform traditional refueling processes, Stratom envisions a future where robot arms can autonomously identify fuel caps and initiate refueling, streamlining operations and reducing human intervention.

Existing Software RevisionsHumble

While previous iterations of software might exist in the domain of computer vision and ML, our project introduced a novel approach by integrating these technologies with the ROS2 [7] Humble framework. By doing so, we brought together the power of robotics and AI, expanding the potential applications of both fields. The need for continued effort arises from the desire to bridge the gap between technology and automotive needs, ensuring the seamless integration of autonomous vehicle systems.

Data Sources and Hardware Interface

In order to train both our bounding box [1] and key points [4] ML models effectively, we required diverse data sources containing images of various fuel cap positions. Our project's hardware interface revolved around the utilization of a RGBD camera, which allowed us to not only capture visual data, but to also incorporate depth information allowing for 3D image recognition. This synergy of visual and spatial data enhanced the accuracy and robustness of our recognition algorithms.

Stakeholders and Maintenance

The stakeholders of our software span a wide spectrum, including Stratom's technical team, automotive engineers, and end users who stand to benefit from the convenience of automated vehicle refueling. Our team was responsible for the initial development, but it was envisaged that Stratom's experts will take over in the software's maintenance and continuous improvement, ensuring its seamless integration into their existing systems.

II. Functional Requirements

1. System shall take input from a RealSense 405 Camera
    a. System shall be developed to accept input in the form of an RGB/depth image

2. System shall output:
    a. Annotated image of the fuel port displaying the position and orientation of the fuel cap
    b. PoseStamped [6] ROS message, representing the position and orientation of the fuel cap
    c. Confidence metric
    d. Inference time for the detection methods, in ms
3. The system shall use one or two perception approaches
    a. Perception approaches shall be ready to be modified and/or replaced by Stratom

III. Non-Functional Requirements

1. System shall be written in C++ or Python
2. System shall use current Stratom softwares to allow for easy integration with existing Stratom systems
    a. System shall be a ROS node [8] written using ROS2 [7] Humble
    b. The system shall be able to run in a Docker container based on a Stratom provided image
    c. System shall make use of the realsense2_camera ROS2 [7] package
3. At least one of the perception approaches shall utilize a ML approach
    a. Software shall not require an abundant amount of training
4. System shall aim to detect the fuel cap within reasonable operating ranges
    a. System shall aim to detect the fuel cap within a 7 cm to 50 cm range
    b. System shall aim to detect fuel cap within 30 degrees
5. (Stretch goal) System shall aim to run efficiently to support uninterrupted autonomous refueling, minimizing downtime
    a. System shall aim to perform inference in less than 500 ms

IV. Risks

Risk	Likelihood	Impact	Risk mitigation plan
Damage to robotic equipment	Unlikely	Major	Extensive virtual testing to ensure the software is as intended before testing with hardware
Incorrect real world image annotations (human error)	Likely	Moderate	Develop robust data processing techniques to handle annotation outliers. Implement data validation and sanity checks to ensure accurate fuel cap detection

Selected ML models may not be able to perform well	Likely	Major	Do preemptive research and test multiple approaches; fail fast for approaches which do not work
Lack of expertise in computer vision	Likely	Moderate	Foster a culture of knowledge sharing and continuous learning. Use Stratom as a main point of contact for support.
Unrealistic Timeline	Unlikely	Minor	Develop a detailed project timeline with achievable milestones. Regularly reassess and adjust timeline based on progress.

V. Definition of Done

Minimal Useful Feature Set:

The following key features must be successfully implemented and thoroughly tested:

1. Autonomous Fuel Cap Detection: One of the developed vision approaches shall accurately detect the position and orientation of a fuel cap
2. Pose Stamp: shall produce a ROS2 [7] message containing the position and orientation of the fuel cap

Client Acceptance Testing:

Before acceptance, Stratom expects us to provide detailed documentation and testing information to ensure the software meets their requirements:

1. Accuracy and Precision: A test plan that quantifies the accuracy of detection using each detection method shall be created, executed, and reported. Stratom will validate the accuracy of the fuel cap detection and their annotated images against a fuel cap in their office.

2. Documentation: Team shall provide system documentation including architecture, bring up instructions, and dependencies.

3. Visit Stratom to perform a system demonstration of software.

Delivery Plan

System code will be provided to Stratom using Github and our documentation on how to get the software running in Stratom's environment. Final software in a docker container will be delivered by December 6th, 2023.

VI. System Architecture

Adapting a computer vision project encouraged the team to design several flowcharts on how to collect data to train a ML model, train that ML model, and finally be able to recognize a fuel cap using a RGBD camera in real time. Figure 1 illustrates the step-by-step procedure for detecting a fuel cap using image data, from data collection to result publication.

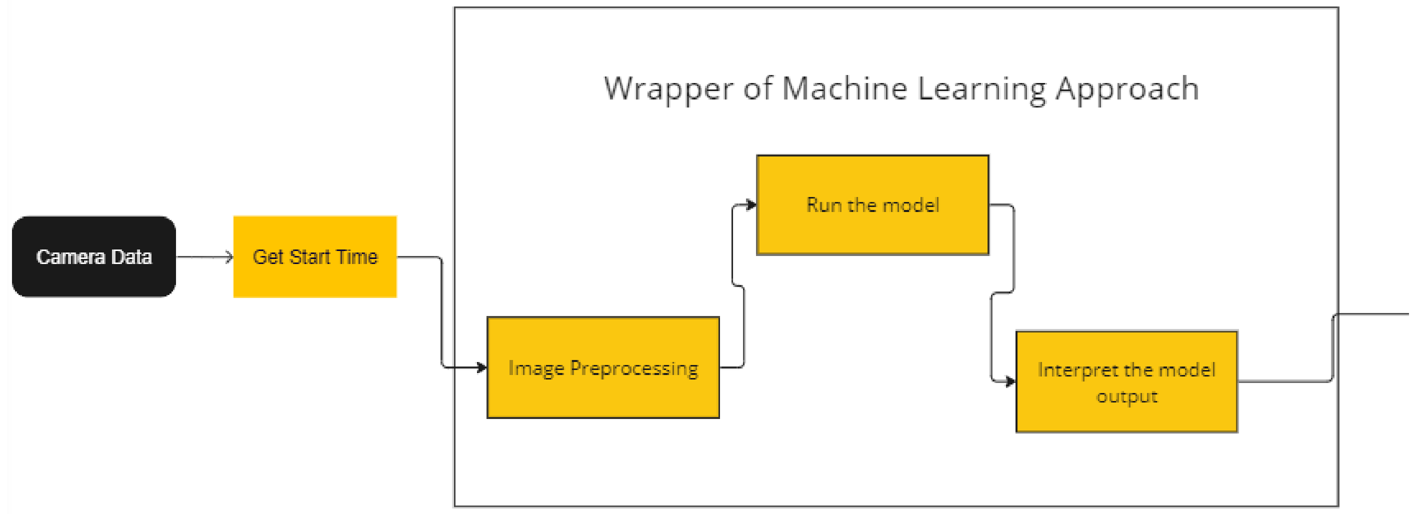

Wrapper of Machine Learning Approach
Camera Data
Get Start Time
Image Preprocessing
Run the model
Interpret the model output

Figure 1: Fuel Cap Detection Process

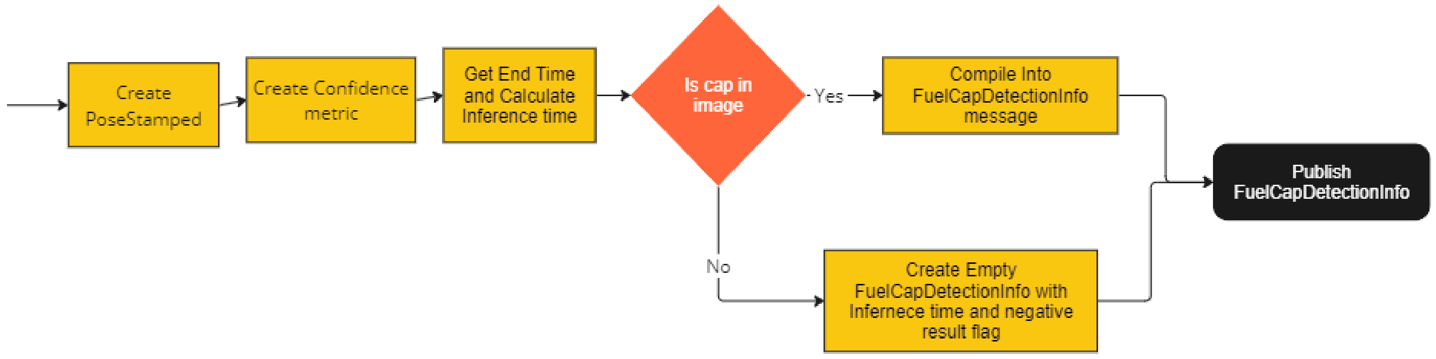

Create PoseStamped
Create Confidence metric
Get End Time and Calculate Inference time
Is cap in image
Yes
No
Compile Into FuelCapDetectionInfo message
Create Empty FuelCapDetectionInfo with Inference time and negative result flag
Publish FuelCapDetectionInfo

Figure 1 (cont): Fuel Cap Detection Process

- At the beginning of the process, a camera is actively collecting data from its surroundings.
- The system logs a start time for calculating an inference time later

Wrapper of ML Approach

- Within this wrapper, the collected image data undergoes several steps:
    - Image Preprocessing: The incoming image data is prepared and normalized for analysis, enhancing its quality and relevance.
    - Run the ML Algorithms: The ML models are executed on the preprocessed image.
    - Interpret the Model Output: The system interprets the output of the ML model, determining the location and orientation of the fuel cap, relative to the camera.
- A 'PoseStamped' [6] ROS message is constructed, representing the position and orientation of the detected fuel cap and the time at which the image was created.
- The system generates a confidence metric that assesses the reliability of the detection results.

- The end time is recorded to calculate the inference time, measuring how long the entire process took.
- Verify that the fuel cap is in the image
    - If a fuel cap is detected, the system compiles the relevant information into a 'FuelCapDetectionInfo' message, including the detected location and orientation.
    - In cases where no fuel cap is found, the system creates an 'Empty FuelCapDetectionInfo' message. This message includes the inference time and a negative result flag to signify the absence of a fuel cap.
- The 'FuelCapDetectionInfo' message is published, making the results available for further processing or display.

Figure 2: Real-World Data Collection Process

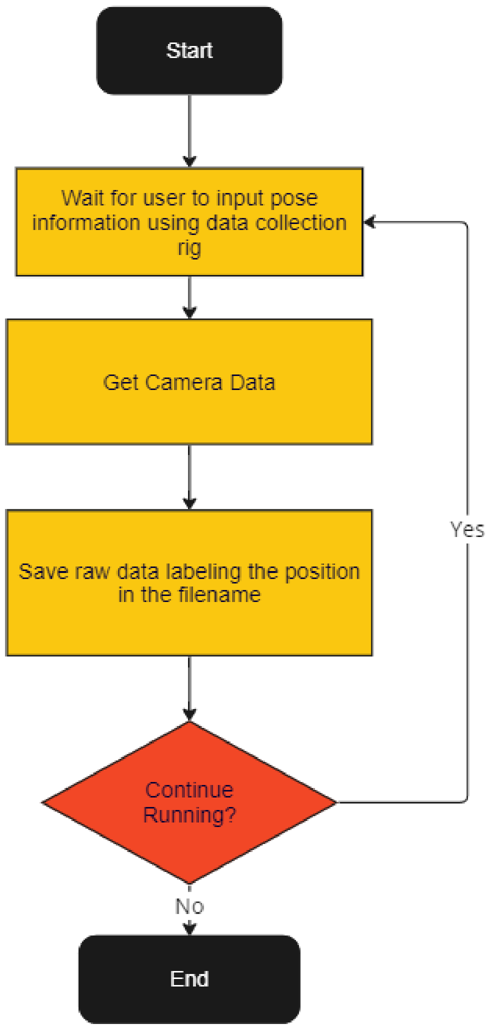

Start
Wait for user to input pose information using data collection rig
Get Camera Data
Save raw data labeling the position in the filename
Continue Running?
Yes
No
End

Figure 2 describes the method used in gathering real-world data for training the ML algorithm. It involves user-defined poses [5], camera data acquisition, and structured data storage, ensuring that the collected datasets are tied to their respective positions for effective ML model training.

1. Wait for User to Input Pose Information

The system waits for user input regarding pose [5] information, which defines the specific spatial characteristics or positions of objects to be captured. This will be done using information collected from a data collection rig to make sure the data is labeled stably and correctly.

2. Get Camera Data

Once the pose [5] information is provided, the system retrieves camera data. This data includes images and possibly additional sensor information, depending on the camera setup.

3. Save Raw Data Labeling the Position in the Filename

The collected raw data is saved, and the system applies a labeling mechanism that embeds the positional information directly into the filenames. This labeling ensures that each dataset is associated with its corresponding pose [5].

4. Continue Running to Collect More Data?

- If further data is required, the process returns to step 1, where the system awaits new pose [5] information.
- When all necessary data has been collected, the process concludes.

Figure 3 outlines a systematic approach to training a ML model, offering the flexibility to create new models or continue improving existing ones. It covers key stages, including model compilation, data preparation, training, performance evaluation, and model preservation for practical application.

Figure 3: ML Model Training Process

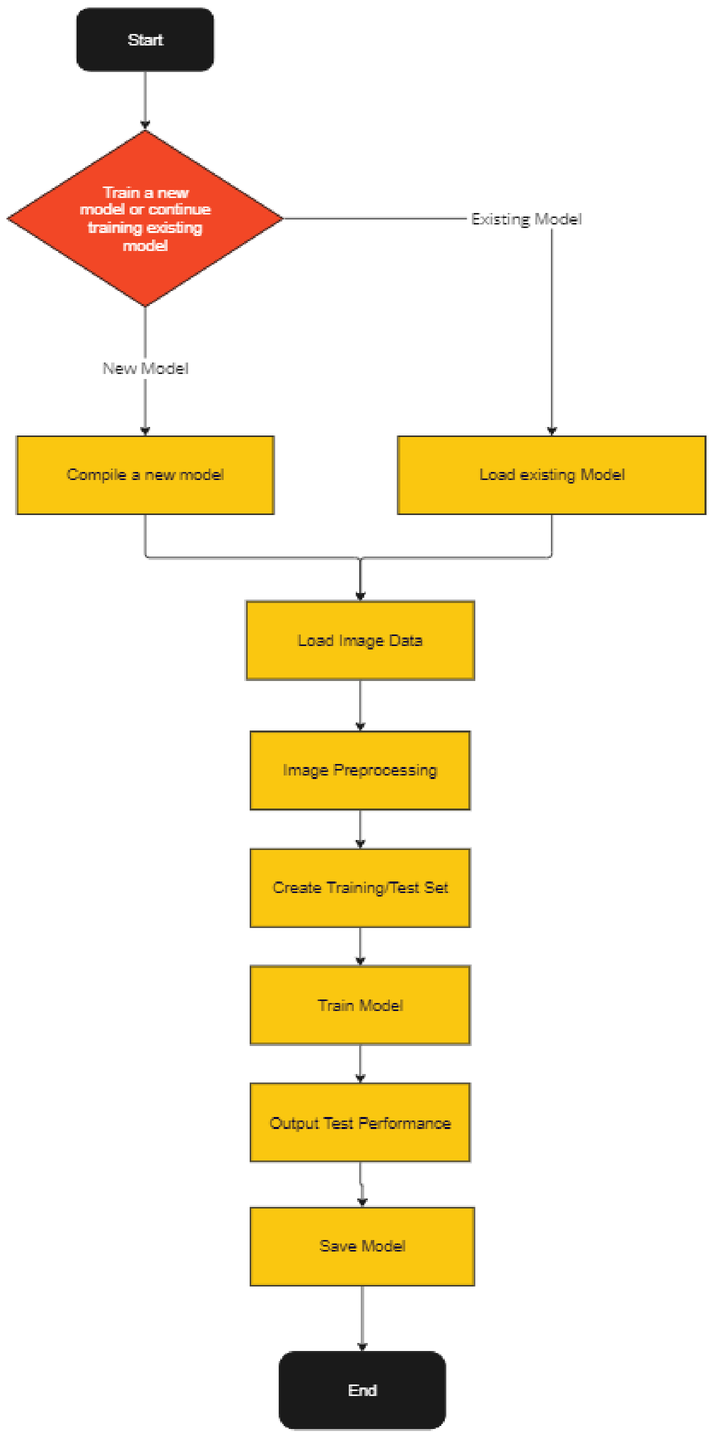
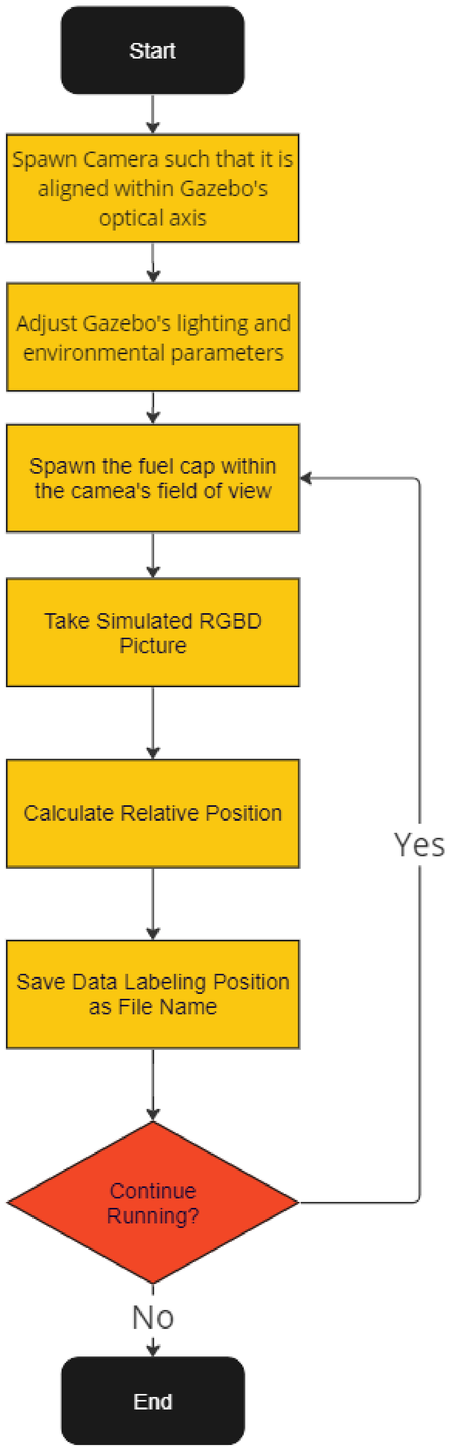


Figure 4: Virtual Data Collection Process


Start
Spawn Camera such that it is aligned within Gazebo's optical axis
Adjust Gazebo's lighting and environmental parameters
Spawn the fuel cap within the camea's field of view
Take Simulated RGBD Picture
Calculate Relative Position
Save Data Labeling Position as File Name
Continue Running?
Yes
No
End

- The process begins by determining whether to create and train a new ML model from scratch or continue training an existing one.
    - If a new model is to be trained, the system compiles a fresh ML model, specifying its architecture, layers, and other configurations.
    - If an existing model is chosen for further training, that previously trained model is loaded into memory to continue the training process seamlessly.

- The system retrieves image data from the dataset, which serves as the foundation for model training.
- The raw image data undergoes preprocessing, which may involve operations like resizing, normalization, and data augmentation to enhance model learning.
- The dataset is divided into distinct training and testing sets, allowing the model to learn patterns from training data and evaluate its performance on testing data.
- The ML model is trained using the prepared training dataset. This involves iteratively adjusting model parameters to minimize prediction errors.
- After training, the model's performance is evaluated on the testing dataset, providing valuable insights into its accuracy and capabilities.
- Once the training is successful, the trained model is saved for future use or deployment.

Figure 4 outlines the team’s approach to gathering synthetic data for training a ML model using the Gazebo simulation environment. It encompasses steps for creating random scenarios, capturing simulated images, calculating relative positions, and saving the labeled data for subsequent model training.

1. Spawn camera such that it is aligned with Gazebo’s optical axis

A virtual camera object is dynamically placed in line with the simulation's coordinates. This will reduce the complexity when reconstructing the fuel cap coordinates in the camera reference frame

2. Adjust Gazebo’s lighting and environmental parameters

We adjust the lighting conditions so that the model will work in a variety of conditions.

3. Spawn the fuel cap within the camera’s field of view

A virtual fuel cap object is instantiated in a location within the simulation so that the camera is pointing at it.

4. Take Simulated RGBD Picture

The virtual camera captures a simulated RGBD image of the scene, mimicking the data acquisition process in the real world.

5. Calculate Relative Position

Based on the captured image and known camera properties, the system calculates the relative position and orientation of the camera with respect to the fuel cap.

6. Save Data Labeling Position as Filename

The collected data, including the RGBD image and calculated relative position, is saved with a filename that reflects the specific position within the virtual environment.

7. Continue Running to Collect More Data?

- If further data is required, the process returns to step 1 for new data collection.
- When all necessary simulated data has been collected, the process concludes.

Technical Design Issues:

1. Problems with Setup:

    - Challenge: Stratom required one of our computers to have the provided dockerfile running on an ubuntu setup. This is necessary for the Docker container to access the USB ports and the RealSense camera drivers. Our team had significant problems setting up the docker file on ubuntu machines.

    - Solution: Communicating and troubleshooting with Stratom allowed us to solve the problem for one Ubuntu machine.

    - Unsolved: Setting up the docker on two machines. This is necessary for efficient data collection.

1. Problems with data collection:

    - Challenge: Receiving accurate camera data is essential for training the machine learning model. Collecting accurate data is a major challenge for the team. The data that must be collected includes a position in 3D space and measurements to notate the pitch, yaw, and roll (rotations about the x, y, and z axes).

    - Solution: Our team designed a method to accurately collect distance, pitch, yaw, and roll.

2. Problems with Gazebo:

    - Challenge: Using the script to use the RealSense camera was not easily translatable to using a simulated camera in Gazebo.

    - Solution: Ditch Gazebo completely and focus on real world data. Simulated data would now be produced using a different software- Unity.

VII. Software Test and Quality

- **Unit Tests**: Unit tests allowed our team to guarantee consistent performance across our codebase. These tests detected and prevented potential bugs. Unit tests made it easier and safer to extend and modify the code ensuring the longevity and adaptability of our product. The aspects of the project that required unit testing include:
    - Data Collection
    - Coordinate Transforms
    - Data Preprocessing
    - Data Post Processing

- **Linux Aliases**: To enhance user experience our team implemented Linux aliases. These shortcuts simplify complex Linux command sequences, enabling users to execute them with just a few key words.

- **Code Reviews**: The team performed code reviews at least once per sprint. The code reviews will be used to ensure our code is well documented, readable and extensible for future users.

- **Confidence interval**: The team created a confidence interval by experimenting with a set of test images and determining the average accuracy of those images. This interval provides users with an idea of how accurate our model is in a given situation, defining its viability.

- **Integration Testing**: The team tested the end-end behavior of data collection, ensuring that it can correctly take in an image and appropriately store and label it. The team also performed integration testing with our machine learning model, making sure the code can take in raw data and give a good pose [5] estimation for that data.

- **Use Acceptance Testing**: The team maintained their code on GitHub in a Docker Container. The team met with Stratom biweekly for demonstrations on code testing as well as allowing Stratom to test the Docker container themselves. Allowing the client to test the team's scripts in the same working space allowed for valuable feedback from the people who have the most domain expertise and hands-on experience in computer vision.

- **Documentation for Code Clarity and Ease of Use:** The code alone may not be self-explanatory. To address this, the team provided in-depth documentation to explain the purpose, functionality, and usage of each ROS node [8] within the codebase. The documentation was written in a clear and understandable manner making it accessible to future developers. The aspects covered include:

    - Explanations of the data collection process
    - Descriptions of coordinate transformation methods
    - Guidelines for data processing, including data formats
    - Processing instructions, highlighting the generation of meaningful results/outputs

VIII. Project Ethical Considerations

Ethical considerations play a pivotal role in the development and implementation of any technological project. In adherence to the principles outlined by the Association for Computing Machinery (ACM), this section delves into the ethical considerations guiding the development of our fuel cap detection project, emphasizing transparency, fairness, and responsible data usage.

ACM Principle: PUBLIC 1.06 - Fair and Honest Representation

Ensuring fairness and avoiding deception are fundamental ethical principles. In accordance with ACM's Principle 1.06, our team maintained transparency in all communications, particularly with stakeholders like Stratom and other users. We committed to:

- Transparent Communication with Stratom: Open and honest communication with Stratom regarding any issues, challenges, or limitations encountered during the project.
- Transparency about Strengths and Weaknesses: Providing clear insights into the strengths and weaknesses of the developed software. This transparency extends to both Stratom and any other entities that may utilize our code.

ACM Principle: PRODUCT 3.10 - Quality Assurance through Testing and Review

Quality assurance is paramount in software development. ACM's Principle 3.10 underscores the need for adequate testing, debugging, and review. In alignment with this principle, our project adhered to the following:

- Unit Testing: Rigorous unit testing is conducted for data collection, preprocessing, and output components, ensuring their accuracy and reliability.
- Model Performance Testing: Thorough testing of model performance is undertaken, accompanied by the provision of confidence intervals to gauge the reliability of the automated refueling system.

ACM Principle: PRODUCT 3.13 - Responsible Data Usage

Ethical data usage is a cornerstone of responsible computing. ACM's Principle 3.13 highlights the importance of using accurate data derived through ethical and lawful means. Our team was committed to:

- Data Collection Practices: Collecting our own data responsibly, ensuring that it adheres to ethical standards and legal requirements.
- Pre-trained Model Usage: Acknowledging the use of a pre-trained model for certain aspects of the project, coupled with a commitment to ethical considerations in its application.

ACM Principle: MANAGEMENT 5.09 - Fair Agreements on Ownership

Fair agreements on ownership are crucial for collaborative projects. ACM's Principle 5.09 emphasizes the need for fair agreements concerning intellectual property. In line with this principle, our team committed to:

- Public Project Sharing: Hosting the project on GitHub as a public repository, fostering transparency and collaboration.
- Fair Software License Agreement: Engaging in discussions with stakeholders, particularly Stratom, to establish a fair and mutually agreeable software license for the project.

ACM Principle: SELF 8.03 - Documentation and Continuous Improvement

Self-improvement and documentation are integral aspects of ethical software engineering. ACM’s Principle 8.03 encourages software engineers to enhance their documentation skills. Our team adhered to this principle by:

- Functional Documentation: Providing comprehensive documentation for the project's functions, ensuring clarity and understanding.
- Self-documenting Code: Employing coding practices that prioritize self-documentation, facilitating ease of understanding for developers and stakeholders.
- Documentation of Common Issues: Creating documentation that addresses common issues, fostering a supportive and informative development environment.

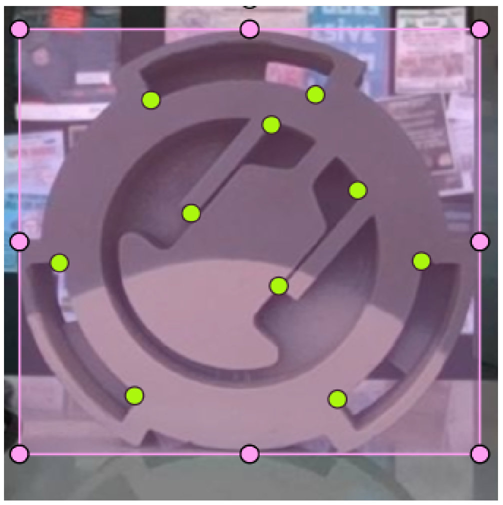
IX. Project Completion Status

Data Collection and Annotation (Real World)

Our team executed a comprehensive real world data collection strategy to amass a diverse dataset for fuel cap recognition. Using a custom Bash script, we set our depth camera to capture 30 frames per second, while an operator moved around the fuel cap, ensuring varied angles and backgrounds were captured. This process yielded over a thousand images featuring the fuel cap against different backgrounds and lighting conditions for a robust data sample.

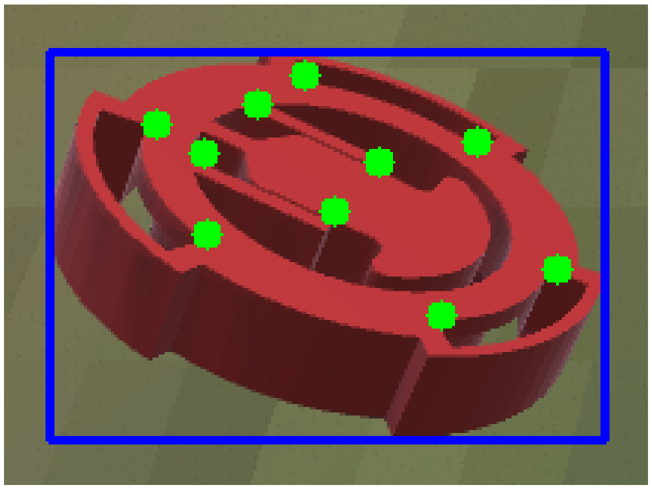
Subsequently, we employed the open data annotation platform CVAT to meticulously annotate approximately 900 images as some of the images were too dark or blurry to use. These annotations included the marking of 10 key points [4] on the front face of the fuel cap and drawing bounding boxes [1] around the cap as shown in Figure 5 by the green points and pink box. The key points [4] were strategically chosen to encapsulate the most critical features defining the fuel cap, while the bounding box [1] outlined the rectangular space framing the fuel cap tightly defining where the fuel cap is in the photo.

Figure 5: Annotated Image (Real World)



Data Collection and Annotation (Simulated)

Our approach to collecting and annotating simulated data was streamlined and highly effective. Leveraging the Unity platform, we crafted a three-dimensional environment replicating the fuel cap's movement and interactions within a camera's perspective. This simulated environment provided controlled scenarios where the fuel cap underwent dynamic rotations, lateral movements, and sporadic reappearances at varying distances from the camera. Each simulated interaction was meticulously designed to ensure a comprehensive and diverse dataset.



Moreover, the annotations for the simulated data were seamlessly integrated into this setup. The 3D model was pre-annotated with key points [4] and bounding boxes [1] that corresponded to the fuel cap's distinctive features as shown in Figure 6 by the green points and blue box. Consequently, any image captured within this simulated environment came pre-annotated, allowing effortless extraction of the key points [4] and bounding box [1] information from a corresponding JSON file.

This methodology streamlined the data generation process, enabling the rapid acquisition of a vast array of annotated images. By blending the precision of simulated environments with predefined annotations, we obtained a high-quality dataset that was pivotal in training our key points [4] ML model.

Figure 6: Annotated Image (Simulated)

Preprocessing

Preparing our dataset for the utilization of a pretrained Faster R-CNN [3] model demanded meticulous preprocessing of our annotated real-world images. Custom Python scripts were developed to ensure data quality by filtering out images that were missing key points [4] or bounding box [1] information, addressing human errors inherent in annotating so many real-world images.

In contrast, the simulated image dataset required a distinct approach. Due to the inherent sharpness of the simulated images, a Gaussian blur filter was applied. This transformation was vital to emulate the real-world scenario where images captured by the depth camera tend to exhibit a certain level of blurriness.

For the key point model training, a series of image manipulations were executed. Initially, images were cropped to isolate only the region encompassed by the bounding box [1]. Standardizing the dataset, the images were then resized uniformly to a resolution of 256x256 pixels. Subsequently, target heat maps were generated, allocating a 10x64x64 structure where each channel represented one of the ten key points [4]. This data was converted into a tensor format and normalized, rendering it compatible with PyTorch for further processing.

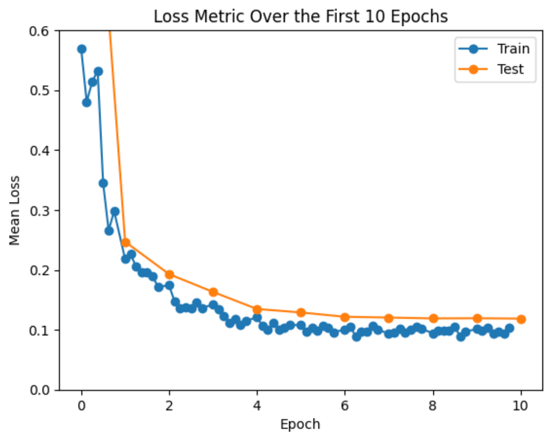
These preprocessing steps played a pivotal role in ensuring data consistency, optimizing the dataset to align with the requirements of the respective models and facilitating the training process.

Bounding Box Model Training

We trained a high resolution Faster R-CNN [3] model with a MobileNetV3-Large FPN backbone to detect bounding boxes [1] for the fuel cap. This model takes an input of a RGB image and for each object it detects it produces an estimate for the bounding box [1] and object class. We initialized this model with weights pretrained on the COCO (Common OBjects in Context) dataset [2], and replaced the object class predictor part of the model with our own class predictor that predicts only whether the object is part of the background or the fuel cap. Bounding box [1] results from this model can be used to focus onto a specific subsection of the image for key point estimation in future work as shown in Figure 8.

The integration of Faster R-CNN [3] into our pipeline marked a significant stride towards achieving accurate and efficient fuel cap recognition. However, since this model takes advantage of only the bounding boxes [1] in the images, and not the image's labeled key points [4], this model can only be immediately used for 3DOF fuel cap position detection.

Figure 7 shows the results of the first 10 epochs of model training. We used a learning rate scheduler which cuts the learning rate in half every epoch, allowing for large gains in performance early on progressing into smaller gains for fine tuning later on. Beyond 10 epochs, we found the model does not improve significantly, regardless of learning rate.



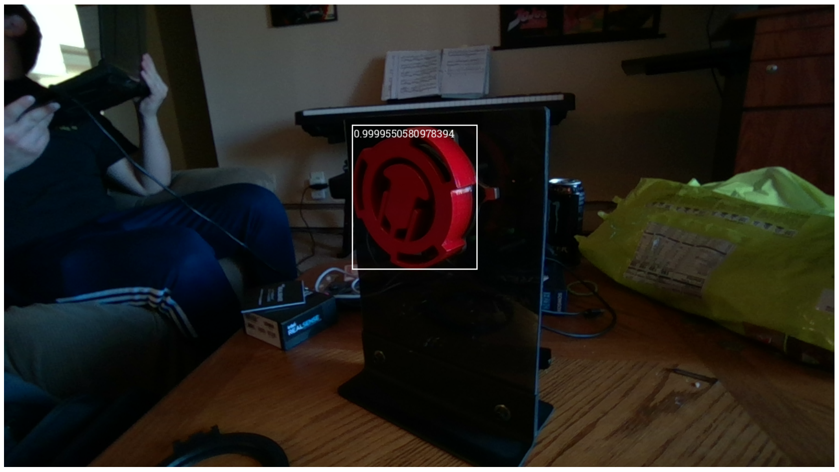


Figure 7: Loss metric of Faster RCNN model over the first 10 epochs

Figure 8: Example bounding box [1] generated from Faster RCNN model

Key Point Model Training

For precise 6-degree-of-freedom (6DOF) pose [5] estimation of the fuel cap, we developed a specialized key point model. This model leverages a sophisticated two-layered hourglass architecture, inspired by the research paper, *6-DoF Object Pose from Semantic key points*, Pavlakos et al.

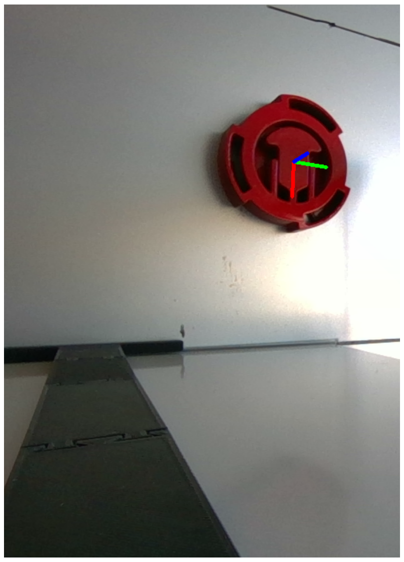
The implementation comprises two distinct stages:

1. Hourglass Architecture: This model configuration consists of two hourglass networks. The primary hourglass network generates initial key point heat maps based on the input image, while the subsequent hourglass network refines the output heatmaps, enhancing their accuracy and clarity.
2. Key point Generation: The final key point prediction is derived by extracting the pixel with the highest value from each channel within the heatmap. These channels correspond to individual key points [4] of the fuel cap, allowing us to precisely pinpoint the cap's spatial positioning in the image.

Coordinate Transformations

The output of our node [8]required us to find a transformation from the camera to the fuel cap. This transformation is what was used to define the pose [5] of the camera, which we defined in a separate python module that we import into our node [8]. Transformations are defined by calculating a simple transformation matrix, described in equation 1, where H corresponds to the 4x4 transformation matrix, R_{3x3} represents a 3x3 rotation matrix, and t_{3x1} represents a 3x1 vector for the translation between the camera to the fuel cap. From this matrix, the orientation was derived from the rotation matrix, while the position is equal to the translation vector. Additionally, given a pose [5] we could build this matrix, which was used for generating an annotated image.

$$H = \begin{bmatrix} R_{3x3} & t_{3x1} \\ 0_{1x3} & 1 \end{bmatrix} \qquad \textbf{(1)}$$



This transformation matrix is built when generating ground truth data, and will be generated by our ML model. Our image annotations are built using this translation matrix to calculate a 3D pose [5], and projected onto an image given our camera's intrinsics matrix. The image annotation is achieved by projecting a coordinate axis from the perspective of the fuel cap onto the image using the intrinsics and transformation matrix defined in the python module as shown in Figure 9.

Figure 9: Example ground truth annotation from module

Implementation

Upon obtaining key point predictions from our trained model, we harnessed this information to ascertain the position and orientation of the fuel cap. To achieve this, we incorporated data from the depth channel in conjunction with the camera's intrinsic parameters to calculate precise 3D coordinates for each predicted key point. By precisely calculating the centroid of eight key points [4] symmetrical about the origin, we could effectively establish the position of the fuel cap within the image. Employing a least squares approach, we calculated the orientation of the fuel cap. This technique allowed us to derive the optimal orientation parameters based on the available information from the key points [4] and their spatial arrangement. With the position and orientation, we could calculate the fuel cap's pose [5] and then generate a PoseStamped [6] message specifically associated with the fuel cap as shown in Figure 10.

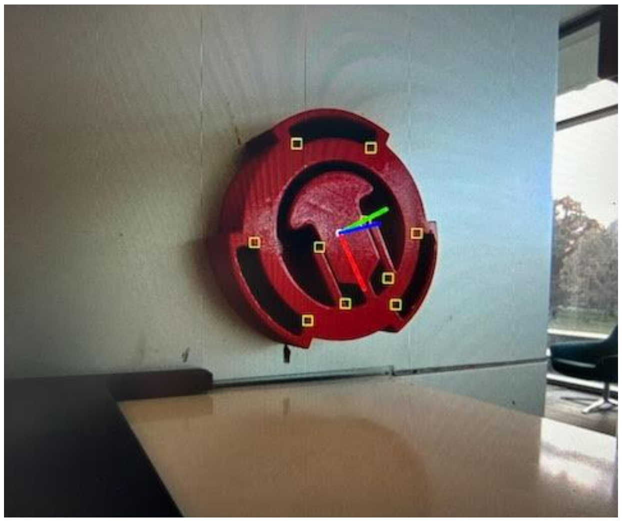


Figure 10: Final Pose Stamped [6] Message

X. Future Work

Our current fuel cap recognition system lays a solid foundation for automated refueling applications. However, there are several avenues for future work and extensions that could enhance the capabilities and robustness of the system.

Improved Key Point Selection:

Refining the selection of key points [4] on the fuel cap could enhance the model's understanding of crucial features such as orientation. Exploring automated methods for key point identification or leveraging advanced techniques like facial landmark detection may provide more accurate and comprehensive annotations.

Dynamic Environment Adaptation:

Extending the model's capabilities to adapt to dynamic environments, such as changes in lighting conditions, diverse backgrounds, and weather scenarios, would be valuable. While we do have some variance in background environments, investigating techniques for domain adaptation and real-time environmental analysis could contribute to increased system resilience.

Continuous Model Training:

Implementing a continuous learning framework that allows the model to adapt and improve over time could also be beneficial. This involves periodically retraining or continuously training the model with new data to account for changes in fuel cap designs, vehicle models, or environmental conditions. This can be done using simulation software such as Gazebo or Unity3D.

XI. Lessons Learned

Technical Proficiency and Continuous Learning:

The project provided an invaluable opportunity for team members, including those with limited prior experience in machine learning and related technologies, to gain hands-on expertise. Delving into computer vision, ROS, and scripting revealed the depth of technical knowledge required for successful project implementation.

Action Over Pure Research:

Emphasizing the importance of hands-on experience, the team recognized that practical application accelerates the learning process. While research and forums offer valuable insights, actively engaging with tools like ROS2 [7] and Gazebo through Docker containers proved to be a more effective and efficient method of mastering these complex technologies.

Real-world Challenges in Industry:

The project simulated real-world industry experiences, where adapting to new software and methodologies took precedence over extensive coding. Navigating the challenges of client-driven projects reinforced the idea that practical problem-solving skills and adaptability are often more crucial than mastery of specific programming languages or tools.

Strategic Decision-Making in Computer Vision:

Working on a computer vision project highlighted the critical nature of decision-making in data preprocessing, model selection, and training. The team learned to navigate the trade-offs between model complexity, dataset size, and training time, gaining insights into the nuanced choices involved in developing effective computer vision solutions.

XII. Acknowledgments

Stratom – Our Valued Client:

Our heartfelt thanks to Stratom for entrusting us with this impactful project. Their accessibility, continuous support, and valuable insights were instrumental in shaping the project's direction. The collaborative spirit exhibited by the Stratom team enhanced our understanding of industry requirements, ultimately influencing the project's outcomes.

Chloe - Our Dedicated Advisor:

Special thanks to Chloe, our dedicated advisor, whose guidance and mentorship played a crucial role throughout the project. Chloe's commitment to our success, timely feedback on presentations, and guidance in adhering to project deadlines were invaluable. Her expertise and encouragement significantly contributed to the project's overall success.

Team Collaboration:

We express gratitude to every team member for their dedication and hard work. Each team member's unique skills and perspectives played a vital role in navigating the challenges and achieving project milestones. The open communication and shared commitment to excellence made this project a fun and productive endeavor.

Academic and Technical Community:

We would like to acknowledge the wider academic and technical community for the wealth of knowledge and resources that significantly contributed to our learning and problem-solving. The forums, tutorials, and documentation provided a robust foundation upon which we built our understanding and implemented various components of the project.

XIII. Team Profile

Xavier Cotton

Senior Computer Science major at the Colorado School of Mines

Jaxon Schauer

Junior Computer Science major at the Colorado School of Mines

David Munro

Senior Computer Science major at the Colorado School of Mines

Keenan Buckley

Senior Computer Science major at the Colorado School of Mines

References

Pavlakos, Georgios & Zhou, Xiaowei & Chan, Aaron & Derpanis, Konstantinos & Daniilidis, Kostas. (2017). 6-DoF Object Pose from Semantic Keypoints [4].

Appendix A – Key Terms

Term	**Definition**
[1] Bounding Box	A rectangular enclosure around an object in an image. Defined by its coordinates, width, and height, providing a visual representation of the object's location and size within the image
[2] COCO Dataset	A large image dataset widely used for object detection, segmentation, and captioning tasks. Contains high-quality images annotated with object instance segmentation, bounding boxes, key points, and object categories
[3] Faster R-CNN	A deep learning-based object detection model that uses a region-based convolutional neural network (R-CNN) architecture, allowing detection of objects within images by predicting bounding boxes
[4] Key points	Refers to specific identifiable points or regions within an image. Often used as reference markers for object detection, recognition, and pose estimation tasks
[5] Pose	Refers to the position and orientation of an object relative to a specific coordinate system
[6] PoseStamped Message	Represents a data structure that combines information about the pose of an object in a specific coordinate frame. Includes timestamped position and orientation data
[7] ROS2	A software that provides tools and libraries for building robot applications
[8] ROS Node	A program that performs specific tasks within a system. Nodes communicate with each other by publishing and subscribing to messages on ROS topics