Remote Visualization and Detection of Foreign Object Debris in Aerospace Manufacturing using a Low-Cost Depth Camera

Ken Latimer

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2019

Reading Committee:

Joseph Garbini, Chair

Santosh Devasia, Chair

Lance McCann

Program Authorized to Offer Degree:
Mechanical Engineering

University of Washington

Abstract

Remote Visualization and Detection of Foreign Object Debris in Aerospace Manufacturing using a Low-Cost Depth Camera

Ken Latimer

Co-Chairs of the Supervisory Committee:
Professor Joseph Garbini
Mechanical Engineering

Professor Santosh Devasia
Mechanical Engineering

Performing work within limited access environments such as aircraft wings is ergonomically hazardous. Often, mechanics are required to crawl through a waist-sized access hole to perform work in a wing. Within these spaces, it is difficult to see and easy to leave behind Foreign Object Debris (FOD). FOD that is left in the aircraft after work is completed can cause expensive damage. Mechanics working in limited access environments are at a high risk of developing Muskuloskeletal disorders [1], and recent FOD issues at Boeing have led to rejection of product deliveries by customers [2]. This research explores remote visualization and automated FOD detection methods as a solution to these problems using a custom-built robot. It was found that remote visualization works well only for debris that is large or distinct in color from the background and that automated FOD detection performs very well when statistics are used to inform selection of the detection threshold. For minimal false positive detections, it was found that at least 40 depth images should be used for each of the initial and final sets of depth images. Detection capabilities were tested in an aircraft wing section using a variety of common debris and it was found that collars as small as 0.27 inches in diameter can be detected, even without visible light.

TABLE OF CONTENTS

LIST OF FIGURES

GLOSSARY

DEPTH IMAGE: An ordered image of depth values

FOD: Foreign Object Debris

POINT CLOUD: Points generated by deprojection of a depth image into space

RVA: Remote Visual Assistant, the prototype robot used in this research

ACKNOWLEDGMENTS

I would like to thank everyone in the BARC lab for their mentorship and assistance during my time in the lab - especially to Cameron Fasola for his help with the mechanical design of the RVA prototype, Cameron Devine for his help with software, and Parker Owan for his ideas about where this research could be taken. Also, I would like to thank Professor Joseph Garbini, Professor Santosh Devasia and Lance McCann for your input and leadership throughout this thesis project.

DEDICATION

To my father, for teaching me the value of hard work and how to turn a wrench - I would not be here today without your guidance and support.

Chapter 1

INTRODUCTION

1.1 Background

Within limited access environments such as an airplane wing, it is difficult to see. Current vision aids such as mirrors and cameras provide some situational awareness, but they are still less than adequate due to their limited field of view. With limited situational awareness, the presence of Foreign Object Debris (FOD) may be difficult to determine. FOD that is not found during manufacturing can cause delays in product delivery as well as expensive damage to components which puts human life at risk.

1.1.1 Current Methods

One method of detecting FOD in aircraft manufacturing, from a Boeing patent, relies on X-ray backscatter to detect changes in a structure. The process involves taking a baseline scan of the structure, facilitating movement of debris, then taking a secondary scan and identifying FOD by subtracting the baseline scan from the secondary scan [3]. The system is able to detect tools, nuts, washers, bolts as well as any metallic and nonmetallic materials left in the aircraft structure [3]. Although this method offers good performance, it is limiting because it requires expensive and proprietary equipment and also requires scans to be taken from the same place.

Another very different FOD detection approach applied to aircraft ground control relies on an actively scanning LiDAR system to detect relatively large FOD on airport aprons [4].

However, this method is not well suited for tight spaces and detection of small debris.

1.2 Motivation

1.2.1 Delays in Product Delivery at Boeing

In February 2019, loose tools and Foreign Object Debris had been found in Boeing's KC-46 tankers – which not only caused deliveries for the United States Air Force to be delayed but also raised a level 3 state of alert at the assembly line in Everett, WA [5]. This is one step away from a level 4 state of alert, which would cause the factory to shut down [5].

Again in March 2019, the Air Force stopped accepting deliveries of the KC-46 tanker due to concerns about FOD. A statement issued from Boeing reads:

> "We have also incorporated additional training, more rigorous clean-as-you-go practices and FOD awareness days across the company to stress the importance and urgency of this issue. Safety and quality are our highest priority." [2]

Boeing is aware of the FOD problem and is actively pursuing solutions. Any way to reduce the amount of FOD that ends up in the aircraft after manufacturing would certainly be beneficial to Boeing.

1.2.2 Ergonomic Concerns in Aerospace Manufacturing

Checking for the presence of FOD can be a physically demanding task. Some operations in aerospace manufacturing require the mechanic to crawl inside of a wing through a small access hole. These restrictive working conditions are likely contributing to the high rates of Musculoskeletal disorders seen in the manufacturing sector [1]. After the mechanic completes their work inside of the wing, it is possible that some FOD was left in the space resulting from the performed operations. Visibility is limited within the wing and the mechanic may not have line of sight to areas that need to be checked for FOD. Subsequently, the mechanic

may need to reorient themselves within the wing or exit the wing and use mirrors, cameras or other vision aids.

1.3 Proposed Solution

The proposed solution is to develop a robot: the Remote Visual Assistant (RVA). The RVA should be compact, inexpensive, able to perceive its environment and small enough to fit through access holes present on aircraft wings. The RVA will offer assistance to aerospace mechanics by:

1. Allowing for remote visualization of limited access spaces, and

2. Automating detection of areas that may contain FOD.

The RVA should be quick to deploy and provide useful visual assistance for mechanics. Then, the RVA is a tool that would be feasible for use in a production setting. See Appendix A for details about the prototype.

1.4 Research Questions

In this research on remote visualization and FOD detection, the following questions are sought to be answered:

1.4.1 Remote Visualization

⟶ **Q1** Is remote visualization sufficient to detect FOD, relying solely on human expertise?

⟶ **Q2** Does the technology developed in this research practically reduce physical strain for the mechanic?

1.4.2 FOD Detection

⟶ **Q3** Does automating the detection of FOD offer an improvement over remote visualization?

⟶ **Q4** For a static camera position, can knowledge of measurement error be used to improve detection performance?

⟶ **Q5** If the camera needs to be moved during the detection process, how well can FOD still be detected?

Chapter 2

REMOTE VISUALIZATION

Giving mechanics access to remote visualization technology helps reduce physical strain that is encountered in their day to day work responsibilities by reducing the need for them to enter the limited access space. In this chapter, two methods of remote visualization are explored - live video streaming and 3D mapping.

2.1 Live Video with Pan and Tilt Control

This mode of operation allows the mechanic to control the pan and tilt axes of the robot with an Xbox controller while seeing a live video stream of the limited access space (Figure 2.1).

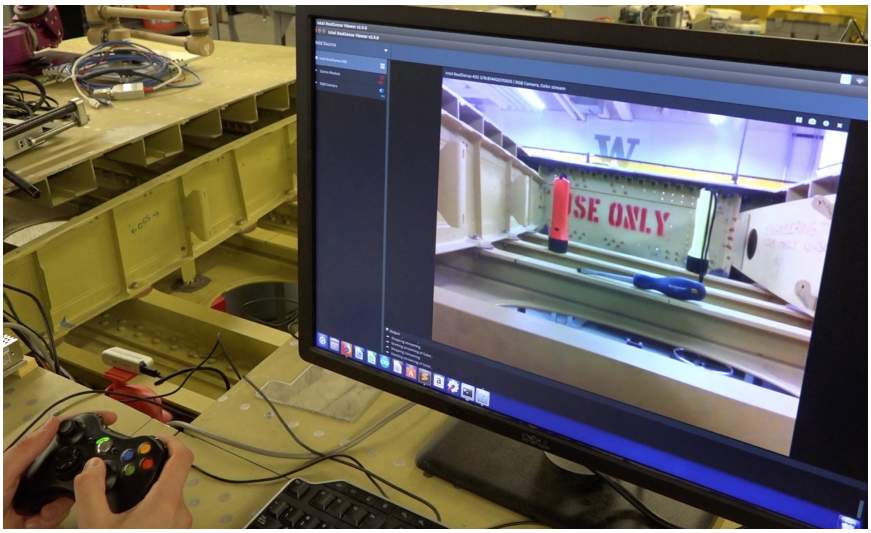


Figure 2.1: Xbox controller operation: the robot is operated remotely with an xbox controller, while a live video stream is observed - allowing for first-person view FOD inspection without being physically present in the wing.

For quick inspections, this reduces the need for the mechanic to enter the wing while also taking less time to deploy in comparison to crawling through the access hole. Although live video streaming is adequate for a human operator to recognize FOD in most cases, some debris in the environment may be difficult to distinguish from a camera image alone. For example, anodized aluminum collars look almost identical in color to the interior of the wing.

2.2 Mapping

Mapping software was implemented which generates a point cloud model of the environment and presents it to the user for inspection. Mapping has an advantage over live video streaming in that it incorporates the physical structure of the scene - allowing for FOD in the wing to be more easily identified.

2.2.1 Theory

Mapping refers to the robotics problem of constructing a map of the environment given sensor data and the robot's pose. In order to form a map of the environment it is necessary to transform point clouds collected in the camera frame into a global frame aligned with the robot's base. For this application, the robot's pose is defined by the joint angles alone since the base is assumed to remain in the same place. Sensor data is comprised of a depth image which is converted to a point cloud at each waypoint in the mapping process.

A point in the camera frame $\mathbf{p}_{\text{camera}}$ can be transformed into a point representation $\mathbf{p}_{\text{world}}$ in the world frame using a transformation matrix $\mathbf{T}_{\text{world}\leftarrow\text{camera}}$ (Equation 2.1).

$$\mathbf{p}_{\text{world}} = \mathbf{T}_{\text{world}\leftarrow\text{camera}} * \mathbf{p}_{\text{camera}} \tag{2.1}$$

A general homogeneous transformation matrix $\mathbf{T}$ consists of a rotation and translation component and is constructed as

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \vec{\mathbf{0}} & 1 \end{bmatrix} \tag{2.2}$$

with a translation vector $\mathbf{t}$ and rotation matrix $\mathbf{R}$ [6] defined as

$$\mathbf{t} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \tag{2.3}$$

$$\mathbf{R} = \mathbf{R}_x\mathbf{R}_y\mathbf{R}_z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_1 & -\sin\theta_1 \\ 0 & \sin\theta_1 & \cos\theta_1 \end{bmatrix} \begin{bmatrix} \cos\theta_2 & 0 & \sin\theta_2 \\ 0 & 1 & 0 \\ -\sin\theta_2 & 0 & \cos\theta_2 \end{bmatrix} \begin{bmatrix} \cos\theta_3 & -\sin\theta_3 & 0 \\ \sin\theta_3 & \cos\theta_3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

where θ_1, θ_2 and θ_3 are the angles of rotation about the x, y and z axes, respectively, and t_x, t_y and t_z are the corresponding translations (Equation 2.3). The rotation matrices in Equation 2.4 are implemented in the CoreRobotics library [7]. The overall camera to world frame transformation matrix $\mathbf{T}_{\text{world}\leftarrow\text{camera}}$ is formed from a series of transformations through the robot's joints.

$$\mathbf{T}_{\text{world}\leftarrow\text{camera}} = \mathbf{T}_{\text{world}\leftarrow\text{servo1}}\mathbf{T}_{\text{servo1}\leftarrow\text{bracket}}\mathbf{T}_{\text{bracket}\leftarrow\text{camera}} \tag{2.5}$$

Each of the transformation matrices $\mathbf{T}_{\text{world}\leftarrow\text{servo1}}$, $\mathbf{T}_{\text{servo1}\leftarrow\text{bracket}}$ and $\mathbf{T}_{\text{bracket}\leftarrow\text{camera}}$ from Equation 2.5 can be calculated using Equation 2.2 with an appropriate rotation matrix and translation vector based on geometry of the robot. $\mathbf{T}_{\text{world}\leftarrow\text{servo1}}$ transforms from the base of the robot through the first servo, $\mathbf{T}_{\text{servo1}\leftarrow\text{bracket}}$ transforms from the first servo to the camera bracket connection at the second servo and $\mathbf{T}_{\text{bracket}\leftarrow\text{camera}}$ transforms from the camera bracket into the camera frame with the positive z axis facing outward from the front of the camera.

2.2.2 Implementation

Algorithm 1 Mapping

```
1: initialize camera, robot, waypoints
2: initialize map as empty
3: for w ∈ waypoints do
4:     robot.moveJoints(w)
5:     color ← camera.getColorImage()
6:     depth ← camera.getDepthImage()
7:     points ← depth.getPoints()
8:     joints ← robot.getJointPositions()                    {position feedback}
9:     T_world←camera ← cameraToWorldTransformation(joints)
10:    points.transform(T_world←camera)
11:    map ← map ∪ points
12: end for
```

Algorithm 1 shows pseudocode for the mapping procedure. The robot moves between a series of waypoints, converting depth images into point clouds and transforming them into the world frame. The transformation is done by constructing a transformation matrix (Equation 2.5) at each waypoint using the current joint positions reported by each servo's position feedback. Points are concatenated into a global set of points which represent a map of the environment.

Figure 2.2: Mapping procedure: the robot moves between waypoints, collecting depth images, converting them to point clouds and transforming them into the world frame to construct a 3D map of the wing.

The mapping procedure from Algorithm 1 was implemented in C++ and used to map the inside of the wing as shown in Figure 2.2. The 3D map shown in Figure 2.3 reveals structure of the scene that a color image alone cannot provide - assisting the operator to more easily

recognize debris. However, the map is still difficult to interpret. In the next chapter, a more interpretable visualization method will be developed.

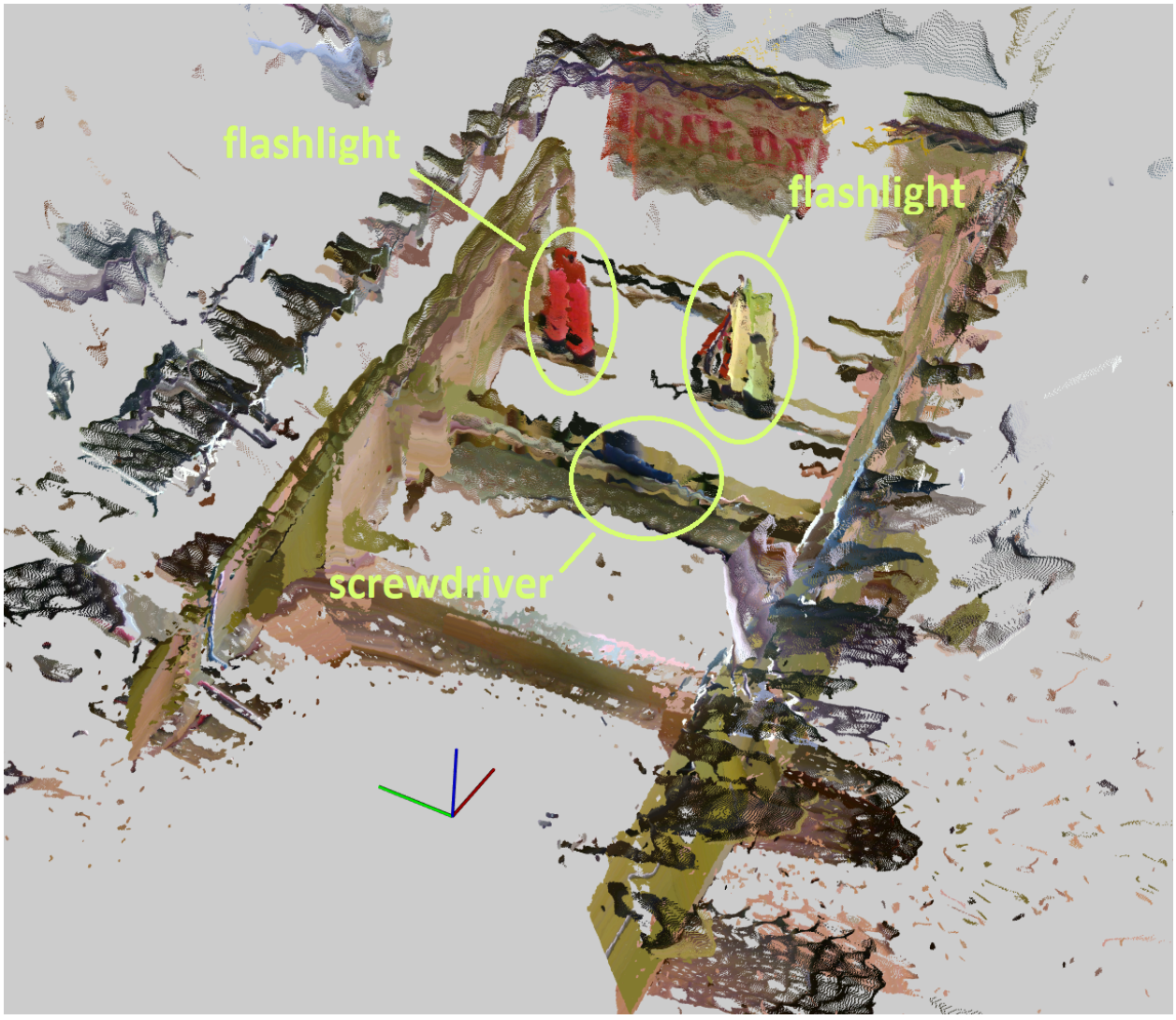


Figure 2.3: 3D Mapping result: Three FOD specimens present in the wing and labeled by hand.

2.3 Summary

The two operating modes for the RVA presented in this section allow for the mechanic to perform inspection for FOD without being physically present in the wing. This reduces physical strain on the mechanic and allows for better visibility in the wing.

Answers to Research Questions (Section 1.4.1)

Q1 Is remote visualization sufficient to detect FOD, relying solely on human expertise?

> Live video streaming is sufficient for relatively large objects that are significantly different in color and shape from their environment, but this method is inadequate for very small debris or debris that looks similar in color to the background. Mapping offers an improvement over live video streaming because the three-dimensional structure of the scene is more apparent. Although these methods improve situational awareness of the limited access space, they still rely on the human operator to recognize FOD and are not free from human error.

Q2 Does the technology developed in this research practically reduce physical strain for the mechanic?

> Although these methods are subject to human error, they do offer the benefit of not requiring the mechanic to enter the limited access space a second time for FOD inspection - thereby reducing physical strain. The mechanic can easily deploy the robot through the access hole with one arm and remain outside of the wing to perform inspection. It is certainly less strenuous for the mechanic to use a remote visualization strategy in comparison to other vision aids or re-entering the wing.

Chapter 3

FOD DETECTION

The previous chapter on remote visualization presented methods to see inside an environment but relies entirely on the human operator to recognize FOD from either a live video stream or 3D map. Now, some automation will be introduced in order to assist the human operator with recognizing FOD.

3.1 Overview

The methods presented in this chapter rely upon the following assumption. Before the mechanic enters the wing to perform a task, an initial dataset is collected. After the mechanic finishes their work and exits the wing, a final dataset is collected. The initial and final datasets are compared to detect which parts of the scene have changed, and the changed areas are presented to the user for further inspection. The changed areas may contain installed work or Foreign Object Debris.

3.2 Depth-based Detection

In addition to the previously stated assumption, this method will assume that the robot's pose does not change between collection of the initial and final datasets. This means that the data can be kept in an ordered depth image and that pixels can be compared directly between the initial and final datasets.

Figure 3.1 shows a flashlight, screwdriver and key placed in the wing. This scenario will be examined in the following sections.

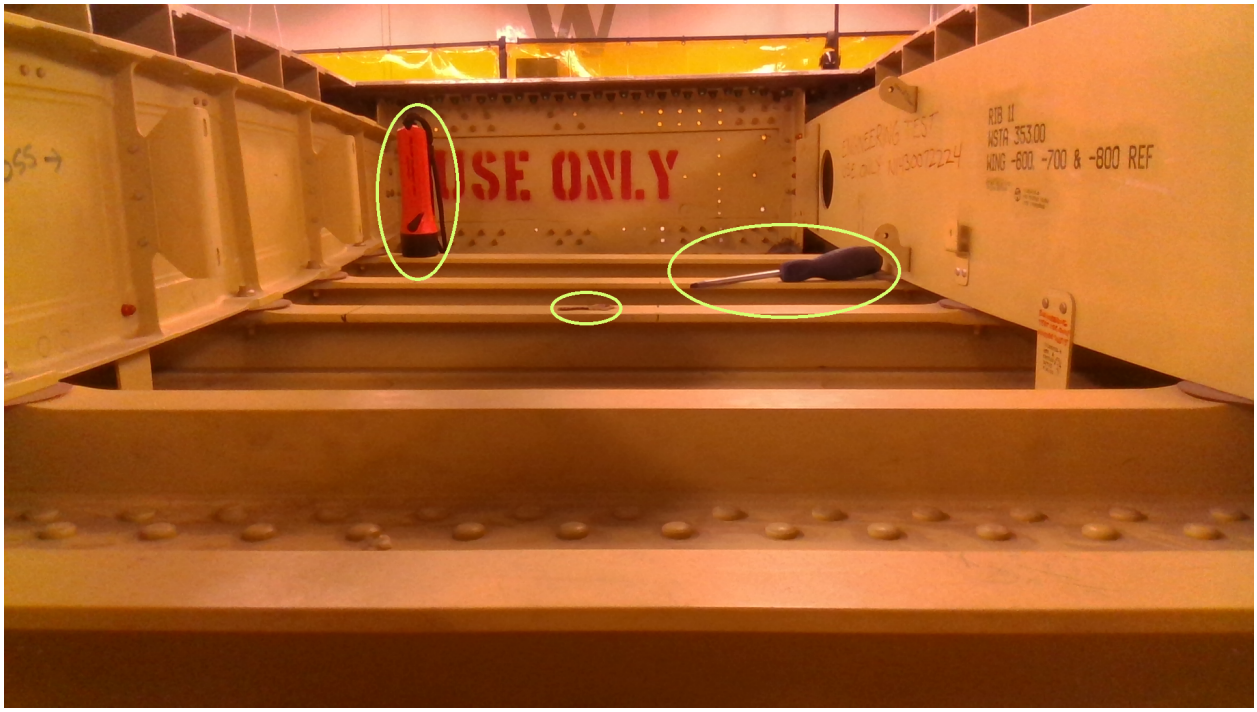


Figure 3.1: Debris placed in the wing: flashlight, key and screwdriver labeled by hand.

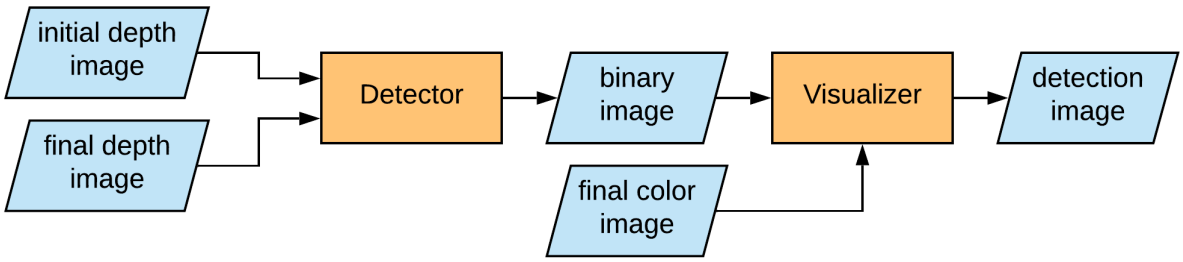


Figure 3.2: Depth-based detection pipeline: initial and final depth images are used to form a binary image where pixels with a detection are true. The binary image is combined with the final color image, containing debris, to present detections to the user in an interpretable way.

3.2.1 Detection Pipeline

A flowchart for the detection pipeline is shown in Figure 3.2. The first significant component is the Detector, which determines areas of the scene that have changed using only the initial

and final depth information. The Detector generates a binary image which is used by the Visualizer to overlay bounding box predictions of FOD locations on top of the final color image which may contain debris.

3.2.2 Constant Detection Threshold

The simplest way to detect changes in the scene is to look for pixels that changed by more than a constant threshold between two depth images (Algorithm 2). Pixels that change by more than the threshold are marked as a detection while ignoring depth values further away than the clipping distance[1]. In this case, the clipping distance is set to 2 meters in order to ignore any detections that would occur outside of the wing.

Algorithm 2 Debris Detection, Constant Threshold

```
initialize threshold, clipping distance
initial ← camera.getDepthImage()
final ← camera.getDepthImage()                          {debris has been introduced}
initialize binary image changes with zeros
for int i = 0; i < initial.resolution.height do
    for int j = 0; j < initial.resolution.width do
        error ← initial(i, j) - final(i, j)
        if error > threshold and final(i, j) < clipping distance then
            changes(i, j) ← 1
        end if
    end for
end for
detections ← drawChanges(final, changes)
```

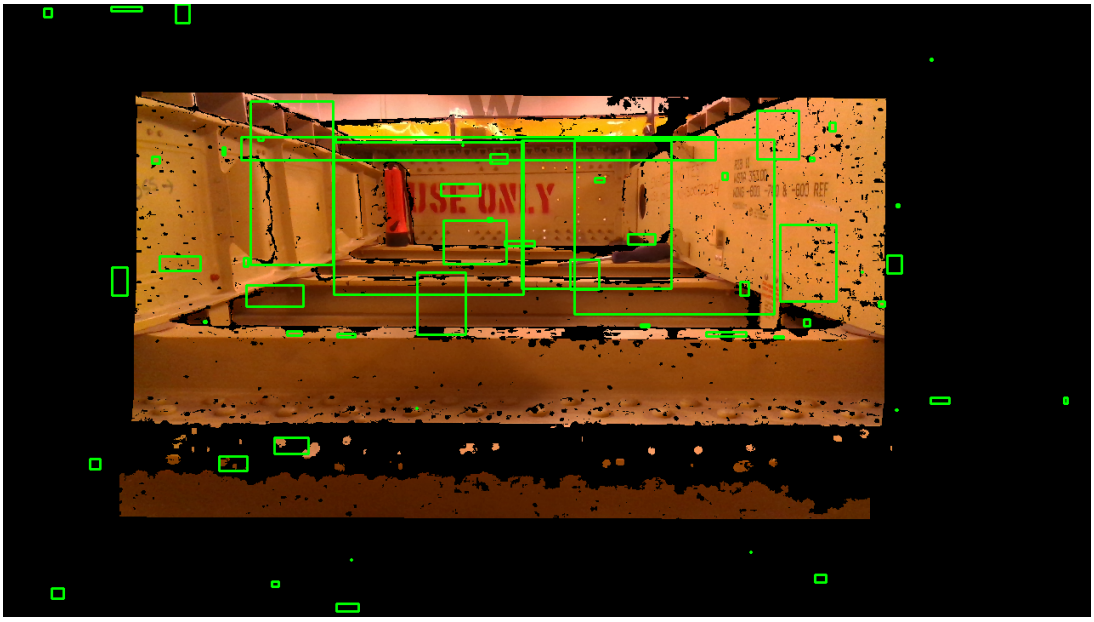
From Figures 3.3 and 3.4 the effect of varying the constant detection threshold can be observed. The detection pipeline is based purely on depth and detections are shown as binary blobs in Figure 3.3. The corresponding detection images are shown in Figure 3.4. See Appendix B for implementation details of the `drawChanges` function from Algorithm 2. Note

[1]See example on depth-based background removal here: `https://github.com/IntelRealSense/librealsense/tree/master/examples/align-advanced`

(a) 1 centimeter threshold

(b) 5 centimeter threshold

Figure 3.3: Binary image - varying the constant detection threshold. This is the output from the Detector stage in Figure 3.2.



(a) 1 centimeter threshold

(b) 5 centimeter threshold

Figure 3.4: Detection image - varying the constant detection threshold. This is the output from the Visualizer stage in Figure 3.2.

that the depth and color images possess a physical offset and different field of view, so depth-based detections may occur outside of the color frame. Setting a threshold of 1 centimeter (Figures 3.3a and 3.4a) results in a true positive detection for the flashlight and screwdriver; however, the swath of false positives hinders interpretability. Increasing the threshold to 5 centimeters (Figures 3.3b and 3.4b) greatly reduces the number of false positives, but the key is still not detected. Using a constant threshold in this way imposes an undesirable and performance-limiting tradeoff.

3.2.3 Noise Characterization

To have the most accurate detections possible, it is necessary to characterize noise within a collection of depth images. To characterize how noise is distributed in the frame, 100 depth images were sampled of an empty wing and the mean and standard deviation were calculated for each pixel in the frame (Figure 3.5). The jet colormap[2] was applied to the standard deviation image using the OpenCV library [8] and placed alongside the image as a reference scale with approximate values labeled. Notice that the majority of the scene has a low standard deviation but there is a noticeable increase around edges and at certain patches in the near field. This indicates that noise depends mostly on the structure of the scene and can be determined on-the-fly for whatever environment in which the camera is placed. Default settings are used for the camera. Intel specifies a minimum depth distance of 0.11 meters, which explains why some of the errors near the bottom of the frame are present.

3.2.4 Gaussian Detection Threshold

Now, 100 depth images are used for each of the initial and final data sets. The constant detection threshold in Section 3.2.2 used only the first depth image from each of the initial and final data sets used here.

---

[2] `https://docs.opencv.org/2.4/modules/contrib/doc/facerec/colormaps.html`

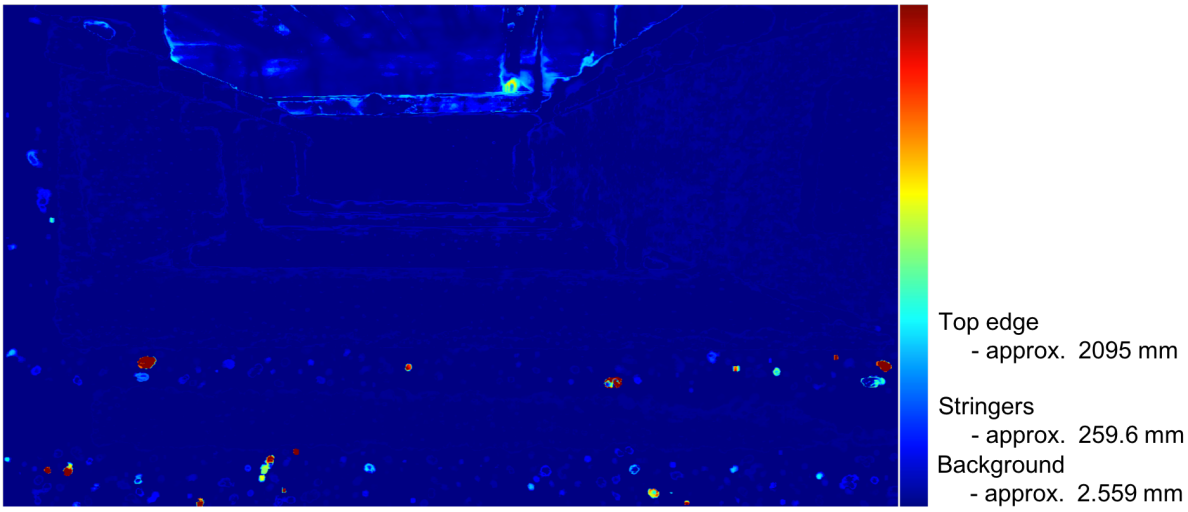


Figure 3.5: Colorized standard deviation of depth values: standard deviation is shown per-pixel for 100 depth images. Values are relatively low for the background and stringers, but increase significantly around the top edge of the wing where there is a large change in depth between the wing interior and back wall of the room.

With knowledge of the mean and standard deviation for each pixel in the depth frame, for both the initial and final datasets, a detection threshold may be set on a pixel-by-pixel basis depending on the desired detection confidence. The advantage to this method is that by using statistics, uncertainty can be quantified and used to intelligently select the detection threshold. The criteria for registering a detection is that $t_f < t_i$ where t_i and t_f are defined as:

$$t_i = \mu_i - 3\sigma_i \tag{3.1}$$

$$t_f = \mu_f + 3\sigma_f \tag{3.2}$$

Here μ_i and σ_i are the mean and standard deviation of the initial set of depth images and μ_f and σ_f are the mean and standard deviation of the final set of depth images. The thresholds t_i and t_f are thresholds for each of the initial and final gaussian distributions. The initial threshold t_i is set $3\sigma_i$ closer to the camera from the mean μ_i (Equation 3.1), while the fi-

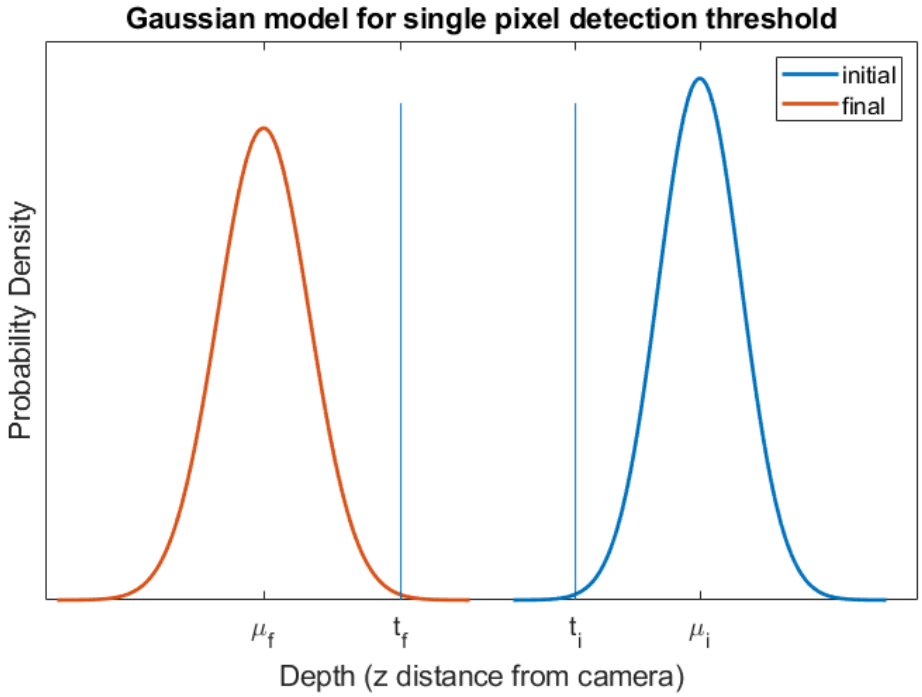


Figure 3.6: Gaussian threshold illustration: if the gaussian models for the initial and final sets of depth images are sufficiently far apart, a detection will be registered for that pixel.

nal threshold t_f is set $3\sigma_f$ farther away from the camera from the mean μ_f (Equation 3.2). Requiring that $t_f < t_i$ to register a detection ensures that the means of the initial and final distributions are separated by a minimum of $3\sigma_i + 3\sigma_f$ for a high detection confidence.

Figure 3.6 illustrates the gaussian threshold model for a single pixel. An assumption is made that $\mu_f < \mu_i$ since any debris in the final dataset will be closer to the camera than the background in the initial dataset. For any pixel where the condition $t_f < t_i$ is satisfied, that pixel is marked as a detection.

The result of applying Algorithm 3 is shown in Figure 3.7. All three items - flashlight, screwdriver and key - are successfully detected using the gaussian threshold with a minor amount of false positives. This result was unable to be achieved with a constant threshold,

Algorithm 3 Debris Detection, Gaussian Threshold

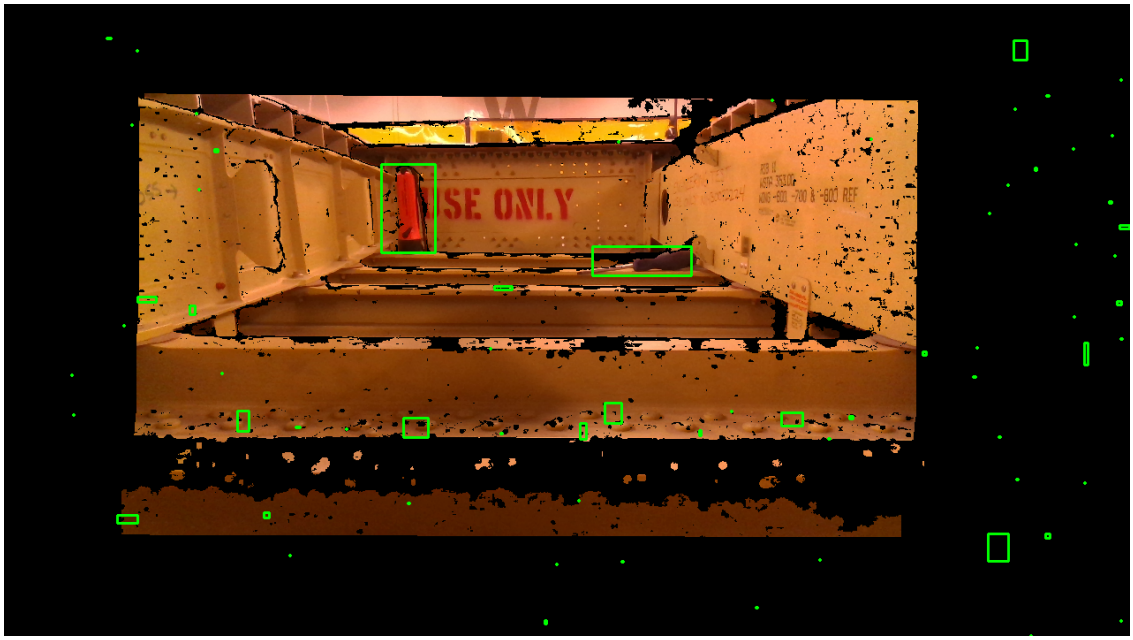
Require: initial and final data contain multiple depth images each.

initialize *camera*
initialize binary image *changes* with zeros
initial ← *camera*.getDepthImages()
final ← *camera*.getDepthImages() {debris has been introduced}
$\mu_i, \sigma_i \leftarrow$ initial.mean(), initial.standardDeviation() {vectorized array calculations}
$\mu_f, \sigma_f \leftarrow$ final.mean(), final.standardDeviation()
$t_i \leftarrow \mu_i - 3\sigma_i$
$t_f \leftarrow \mu_f + 3\sigma_f$
changes ← 1 **where** $t_f < t_i$
detections ← drawChanges(final, *changes*)

Figure 3.7: Detections using a gaussian threshold: All FOD specimens are detected - flashlight, screwdriver and key. This is an improvement over the constant detection threshold in Section 3.2.2 where the key was not detected.

where the key was not detected (Figure 3.4). Notably, most false positives are on the order of one to three pixels which was not the case for the constant threshold.

3.2.5 Number of Depth Images and Detection Quality



(a) 10 depth images

(b) 50 depth images

Figure 3.8: Varying the number of depth images - gaussian threshold

In Section 3.2.4, 100 depth images were used for each of the initial and final datasets because of the low number of false positive detections. It was found that as the number of depth images increases, the number of false positives is reduced at the cost of increased runtime. Figure 3.8 shows the effect of varying the number of depth images, using 10 depth images (Figure 3.8a) and 50 depth images (Figure 3.8b) which may be compared to the results using 100 depth images in Figure 3.7.

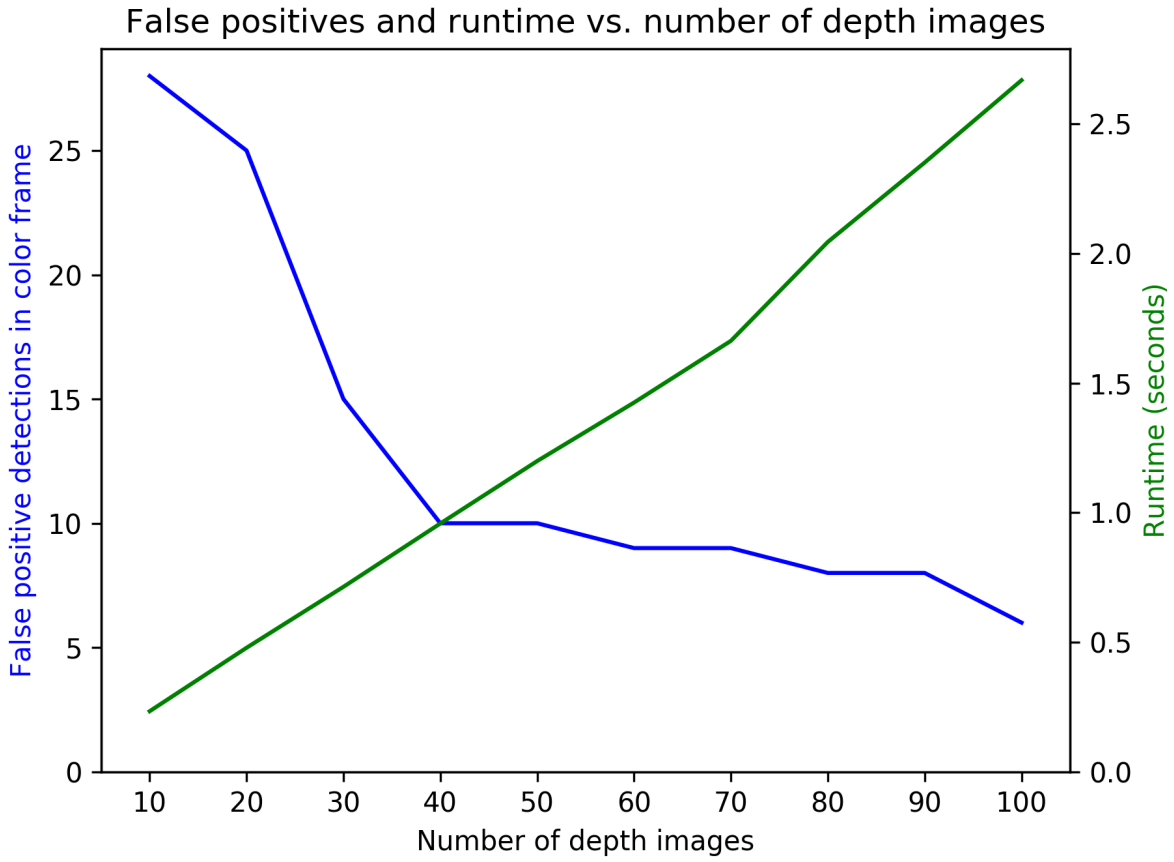


Figure 3.9: Gaussian threshold - detection quality and runtime. As the number of depth images used increases, runtime increases linearly while the number of false positive detections in the color frame asymptotically approaches zero.

Figure 3.9 quantifies how the number of false positive detections and runtime change with the number of depth images used. Detections that occurred in the black border outside of the color camera's field of view were ignored for this plot. The number of depth images used, which is for each of the initial and final datasets, is increased from 10 to 100 in increments of 10. This plot was generated on a base model 2015 MacBook Pro. Notice that as the number of depth images increases, the runtime increases linearly while the number of false positive detections asymptotically approaches zero. This relationship allows for the number of depth images to be selected based on the desired accuracy and speed, which is important for a production environment where time is limited.

3.2.6 Testing with Common Aerospace Debris

Thus far, fasteners are a class of debris that have not been considered. A typical wing manufactured at Boeing could contain thousands of swaged collar fasteners. Four sizes of collars are shown in Figure 3.10 with their outer diameter labeled.

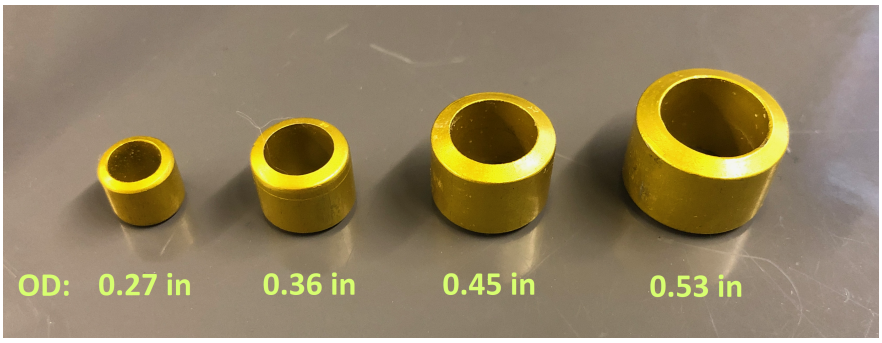
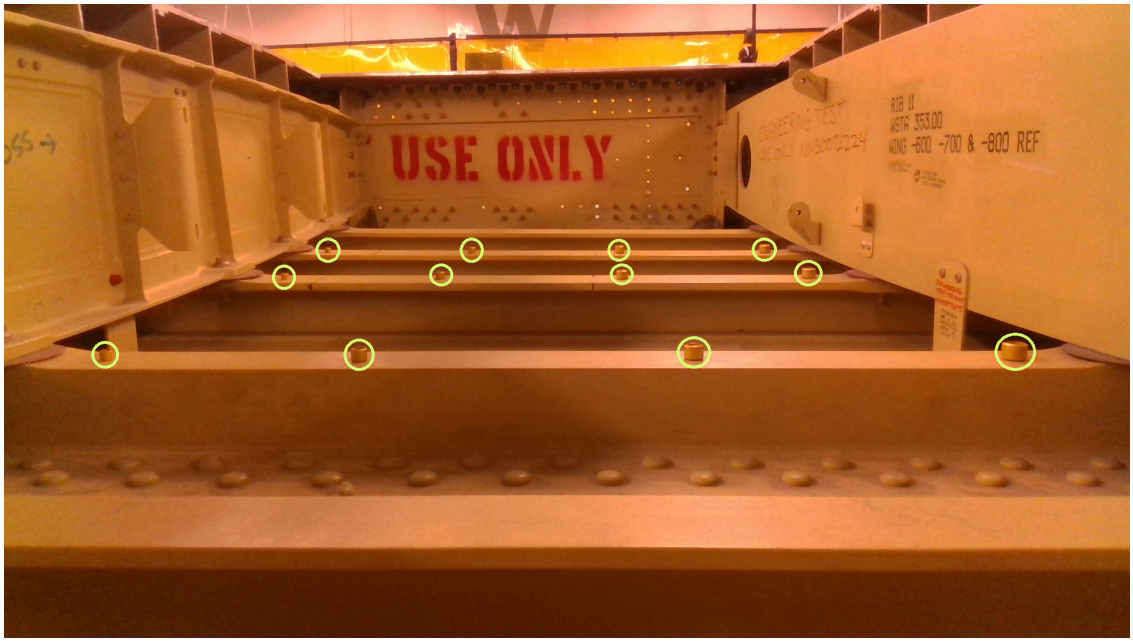
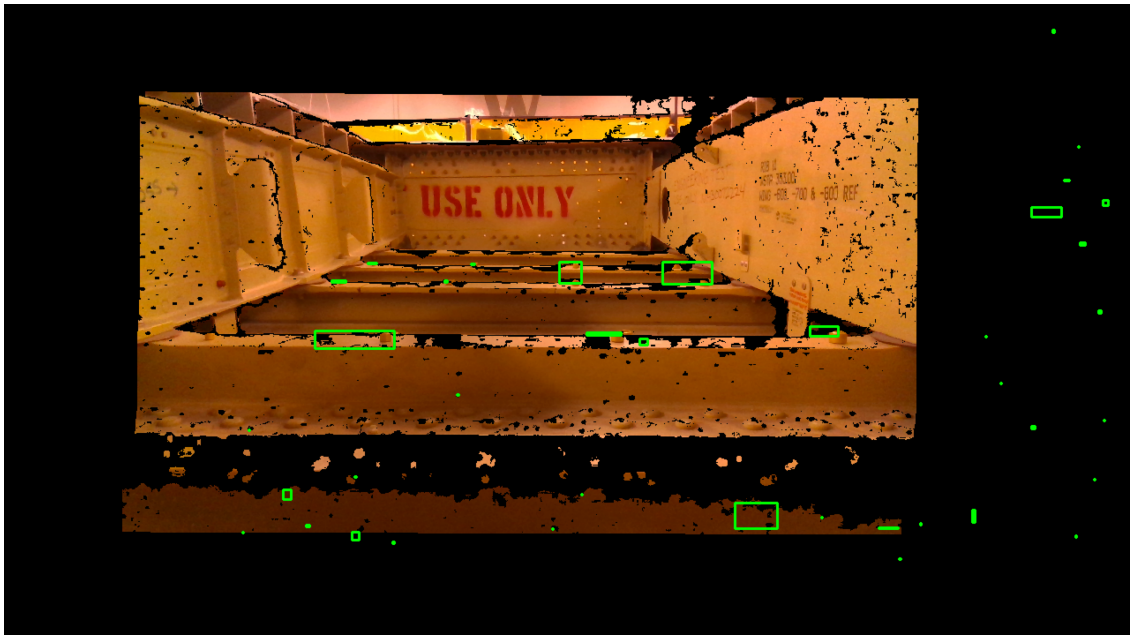


Figure 3.10: Collars - outer diameter labeled

Three rows of four collars were placed in the wing (Figure 3.11). The collars increase in size from about 0.27 inches to 0.53 inches outer diameter, going from left to right along each of the three stringers in the middle of the frame (Figure 3.11a). It can be seen that all of the collars are detected except for one of the smallest collars in the front left (Figure 3.11b). In this test, eleven out of twelve collars were successfully detected.



(a) True positions labeled



(b) Detections

Figure 3.11: Various sizes of collars placed in the wing

(a) True positions labeled

(b) Detections

Figure 3.12: Smallest collars placed along leftmost rib

Finally, three of the smallest collars with 0.27 inch outer diameter were placed in the wing along the leftmost rib (Figure 3.12). The positions of the collars are labeled in Figure 3.12a. Only one out of three collars were detected, even though all three were detected in the previous test (Figure 3.12b). This could be explained by the fact that the infrared dot pattern projected by the depth camera is relatively coarse - so, small objects lying between dots may not be reliably picked up by the depth camera.

3.2.7 Detection without Visible Light

The wing section used in this work has a large cutout on the top of the skin that allows for easy access to the interior of the wing for experimentation. However, a wing in production would be completely sealed except for the access holes. Even without visible light, the detection pipeline would still work because the camera has a built-in infrared projector. In this situation it would not be of much use to see bounding boxes overlaid on a black image. A way to provide contextually meaningful information in this case is to overlay detections on a colorized depth image. The result is shown in Figure 3.13 using a gaussian threshold on the same dataset as before.

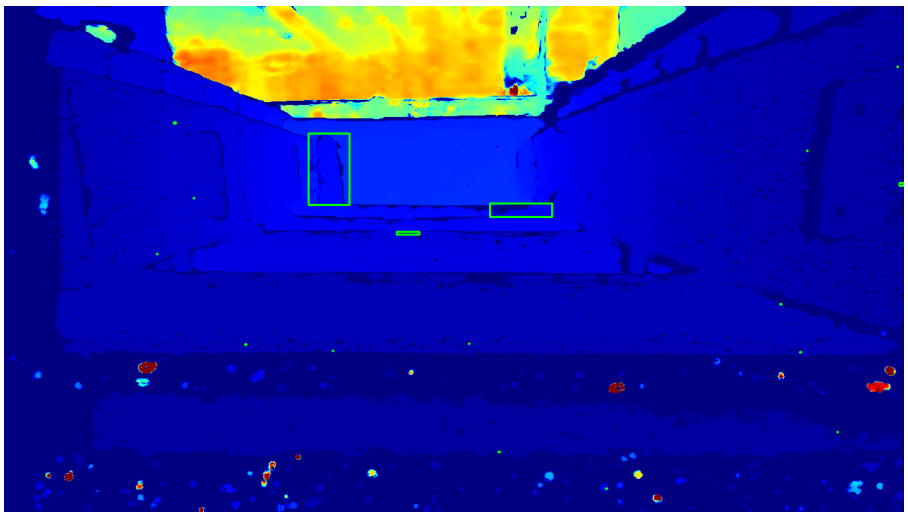
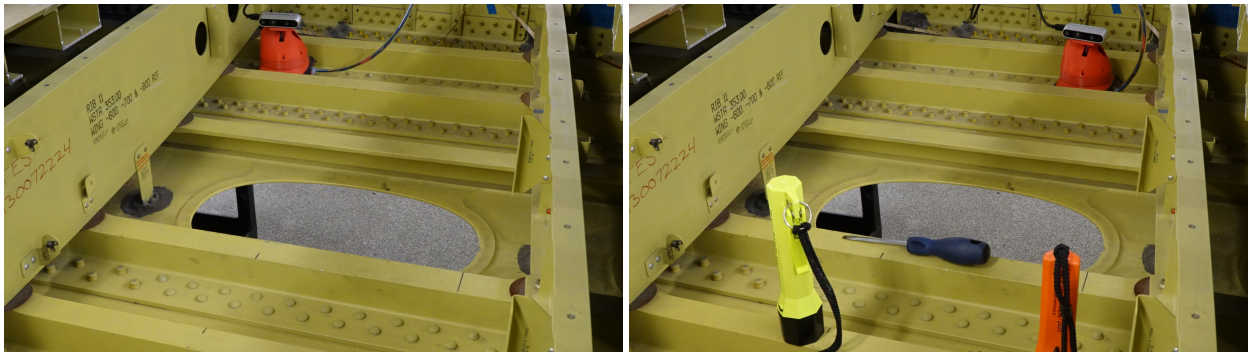


Figure 3.13: Colorized depth image with detections: flashlight, key and screwdriver detections shown with bounding boxes.

3.3 *Spatial Detection*



(a) initial scan (b) final scan

Figure 3.14: Spatial detection experiment: robot is placed in the empty wing, then moved after introducing FOD.

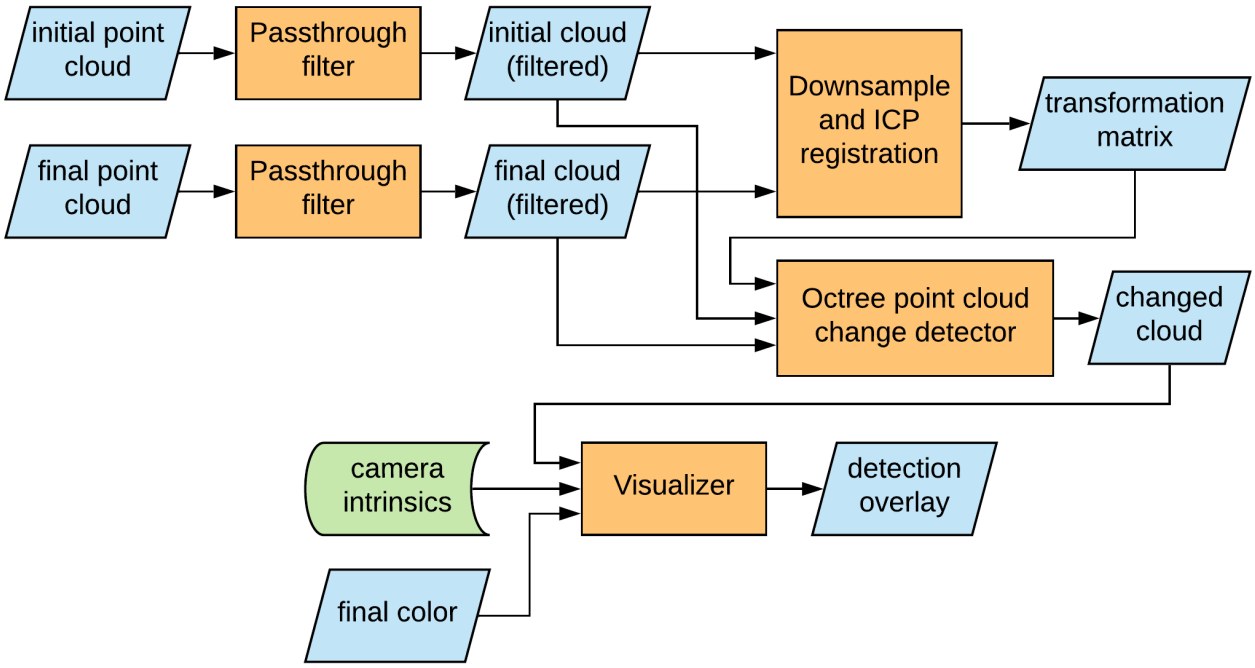


Figure 3.15: Spatial detection pipeline: initial and final point clouds are filtered and aligned before detecting changes and projecting changed regions back into the camera frame.

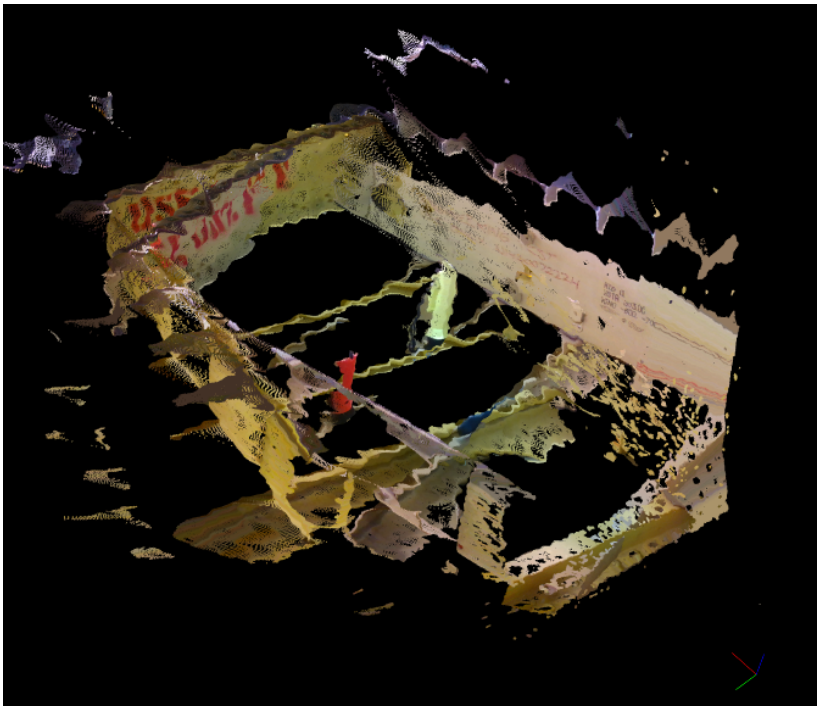


Figure 3.16: filtered clouds

Spatial Detection refers to detecting FOD from a point cloud. The point cloud is generated using a depth image and a camera model to deproject a depth image into space. This is a similar process to the constant detection threshold presented in Section 3.2.2 since a fixed detection resolution is set and a single point cloud is used for the initial and final datasets, but allows for the robot's pose to change in between. This is important if the mechanic needs to perform work exactly where the robot was initially located. The experimental process is shown in Figure 3.14. In this section, two flashlights and a screwdriver are used as FOD specimens. The robot is first placed near the rightmost rib in the camera frame, the initial point cloud is collected (Figure 3.14a), then the robot is moved to the leftmost rib before the debris is introduced and the final point cloud is collected (Figure 3.14b).

The detection pipeline is shown in Figure 3.15. The initial and final point clouds are processed with a Passthrough filter, which is implemented in the Point Cloud Library (PCL) [9],

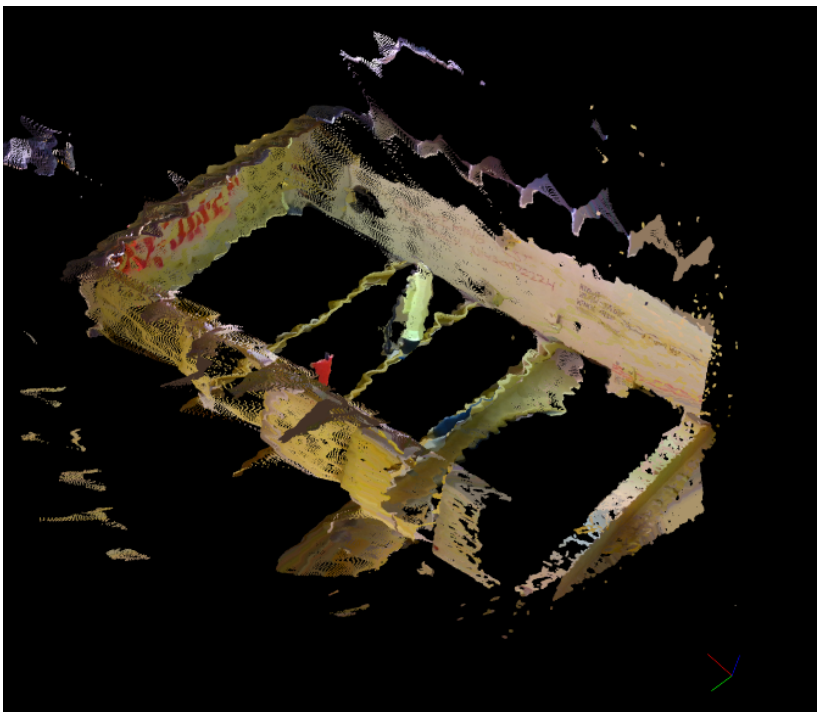


Figure 3.17: aligned clouds

Figure 3.18: changed cloud

shown in Figure 3.16. The Passthrough filter removes points beneath the robot's base and points on the ceiling and back wall that are not relevant for detection. Then, the Iterative Closest Point (ICP) algorithm, also implemented in PCL, outputs a transformation matrix which is used to line up the initial cloud to the final cloud (Figure 3.17). The ICP algorithm works by iteratively selecting a pair of points, one from each of the initial and final clouds, and trying to align them - which is repeated until a termination criterion is reached [10]. Here, PCL's IterativeClosestPointNonLinear implementation is used with 20 alignment iterations, incremental reduction of the maximum correspondence distance, and no termination criteria since the registration converges to a solution after less than 20 iterations. This is based on PCL's tutorial[3] "How to incrementally register pairs of clouds." The maximum correspondence distance is the largest acceptable distance between a pair of points for them to be considered for alignment. This is reduced during each ICP iteration if the previous difference in transformation is less than the current - which indicates that the ICP algorithm is close to convergence and only points that are very close to each other should be considered for alignment. The nonlinear ICP variant is different in that it uses a Levenberg-Marquardt nonlinear optimization algorithm which provides a slightly larger basin of convergence and can find an optimum with slightly reduced error [11].

The filtered and aligned clouds are passed into the Octree point cloud change detector which internally uses an octree data structure to efficiently store spatial information of the two clouds. Here, a voxel grid resolution of 5cm is used for the change detection in order to offer comparable performance to the results shown in Figure 3.4b. By comparing the tree structure of the initial and final clouds, spatial changes can be detected (Figure 3.18). This is based on PCL's tutorial[4] "Spatial change detection on unorganized point cloud data." The changed cloud, which represents detections and is a subset of the final cloud, is combined with camera intrinsics and the final color image to highlight detections in the scene. Camera

[3] `http://pointclouds.org/documentation/tutorials/pairwise_incremental_registration.php`

[4] `http://pointclouds.org/documentation/tutorials/octree_change.php`

intrinsics are the parameters internal to the camera that allow points in 3D to be mapped back to 2D image coordinates. The final image with detections is then presented to the user for inspection.

Figure 3.19: Spatial detection: points from the changed cloud in Figure 3.18 are projected back into the camera frame and the nearest pixel for each projected point is changed to green. All non-green pixels are selectively set to grayscale in post-processing to better visualize the detections.

Figure 3.19 shows the result of projecting the changed cloud into the camera frame. Each point in the changed cloud is projected into the camera frame and the nearest pixel is set to green. The image was originally in color, but was post-processed in photo editing software to set all non-green pixels to grayscale in order to better visualize the detections. Notice that only the two flashlights are detected and not the screwdriver. This is an interesting side effect of using spatial change detection, which internally uses a voxel grid to detect changes. In this test, the screwdriver appears to lie within a series of voxel cells that were already

occupied by the stringer on which it sits. Since those voxels already contained points in the initial point cloud from the stringer, introducing a screwdriver into those voxel grid cells does not trigger a detection. It is a similar case with the bottom part of the flashlights, so detections are not triggered close to the stringer that was present in the initial point cloud. The large patches of detected points on the left side of the image are due to a new part of the scene being exposed after the robot was moved to the left. There is a small amount of left-biased offset present with the overlay, which could be attributed to small errors in coordinate transformations throughout the detection pipeline or errors in the camera projection model.

3.4 Summary

Answers to Research Questions (Section 1.4.2)

Q3 Does automating the detection of FOD offer an improvement over remote visualization?

> Automating detection of FOD offers a significant improvement over live video streaming and 3D mapping by highlighting areas that are likely to contain FOD and not relying on the operator to manually inspect the entire wing. Depth-based detection with a gaussian threshold is able to detect very small debris which is otherwise difficult to distinguish from a color image alone. The yellow anodized aluminum collars are very difficult to see in the color image, but are more easily seen with automated detection.

Q4 For a static camera position, can knowledge of measurement error be used to improve detection performance?

> Using statistics gathered from multiple depth images allows for detection of very small debris in low noise areas of the image, which is a significant improvement over the constant detection threshold.

Q5 If the camera needs to be moved during the detection process, how well can FOD still be detected?

> FOD is not detected nearly as well using spatial detection because the pipeline is less refined. It suffers from the same limitations as the constant detection threshold because noise in the data is not considered. Although, with further refinement, this method shows promise and provides more flexibility in placement of the robot. FOD must be sufficiently large in order to be detected.

Chapter 4
CONCLUSION

4.1 Summary

The primary motives for this research were to 1.) address ergonomic concerns for mechanics by exploring remote visualization techniques and 2.) develop a practical and inexpensive method to identify FOD within the aerospace manufacturing process.

Remote Visualization

Remote visualization techniques including live video streaming and 3D mapping were explored. Although live video streaming is sufficient for relatively large and easily identifiable objects, it is inadequate for small debris or debris that blends in with the background. 3D mapping provides structural information which helps reduce ambiguity from color alone. However, both of these methods are limited by relying on human expertise to recognize FOD. Additionally, a real wing in manufacturing may possess insufficient interior lighting to rely only on a color camera. Nonetheless, these methods reduce the need for a mechanic to crawl into the wing for FOD inspection and thereby reduce the ergonomic risks associated with inspection of limited access spaces.

Detection of Foreign Object Debris

Automating the process of detecting FOD lessens the need for the mechanic to manually inspect the remote visualization results for the presence of FOD. Using a static camera position, two detection methods were explored - constant thresholding and gaussian thresholding. Although using a constant threshold is simple to implement, it suffers from poor detection accuracy and lacks knowledge about the underlying limits of the camera's detection capa-

bility. Using a gaussian threshold overcomes these issues by locally informing each pixel's detection threshold based on the statistics of multiple depth images. The gaussian thresholding technique is able to detect both large and small FOD regardless of lighting conditions and structure of the scene. Spatial detection was developed to allow movement of the camera during the detection process. This technique is unrefined and suffers from the same issues as the constant threshold; however, the added flexibility may be required in some situations and offers decent performance for relatively large FOD. Statistical techniques applied to spatial detection may offer a significant performance improvement as was seen with depth-based detection.

4.2 Future Work

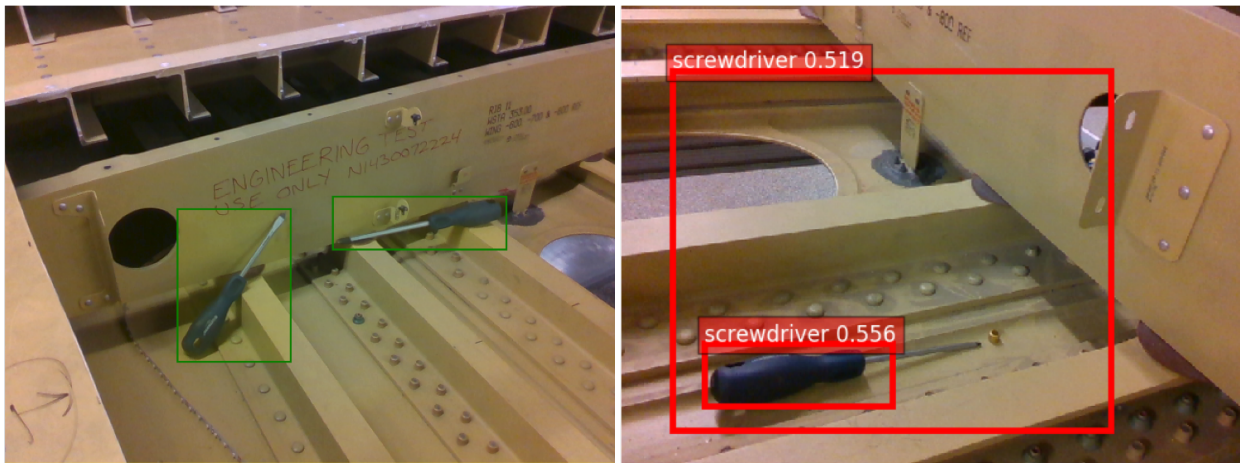
4.2.1 Object Classification

All of the research on detection presented thus far has been focused on labeling regions that contain FOD. The next step in this process would be to segment out the bounding box region proposals and run them through a 2D object detection network such as YOLO [12] in order to classify the detections. Then, the mechanic could be presented with a prediction of where FOD is located as well as what kind of FOD it is. A challenge with applying object detectors like YOLO in aerospace manufacturing is that most are trained on datasets that do not include debris specific to aerospace. This necessitates the need to either gather a large dataset of custom debris for training data, or collect a small dataset of new object classes and finetune an existing network with transfer learning. Preliminary results are shown in Figure 4.1, where a dataset of approximately 50 images with screwdrivers were labeled by hand (Figure 4.1a) using ImgLab[1] in order to finetune a Single Shot MultiBox Detector (SSD) network [13] which was available in the GluonCV library[2]. The output with bounding box predictions and class probabilities is shown in Figure 4.1b. An extension to this could utilize

[1] `https://imglab.ml/`

[2] `https://gluon-cv.mxnet.io/index.html`

3D object detection networks that operate on point clouds such as PointNet [14] to improve classification accuracy. Since a depth camera was used in this research, existing hardware could be used to incorporate classification based on point clouds.



(a) Hand-labeled bounding boxes (b) SSD output

Figure 4.1: SSD for object detection and classification

4.2.2 Effect of Detection Overlays

Although putting a bounding box around debris makes it easier to spot, this also distracts the user from looking at unlabeled areas that they may have otherwise inspected. Further study is required to quantify this effect within the context of aerospace manufacturing.

4.2.3 Depth Sensor On-board Processing

The Intel RealSense D435 was left on default settings throughout this research. However, it is possible that Intel's on-board processing makes adjustments to the depth data over time. Figure 4.2 was generated by performing Welch's Test between the first 10 and last 10 of 100 depth images that were captured sequentially in the empty wing. White pixels are pixels that have a p-value of less than the alpha value of 0.01. Because no debris was introduced, it is not expected to see so many regions that are significantly dissimilar. It may be worthwhile

to disable all of the depth camera's on-board smoothing and post-processing to verify if the camera is making adjustments. This result may help explain some of the false positives in the frame from the depth-based detection techniques that were explored.

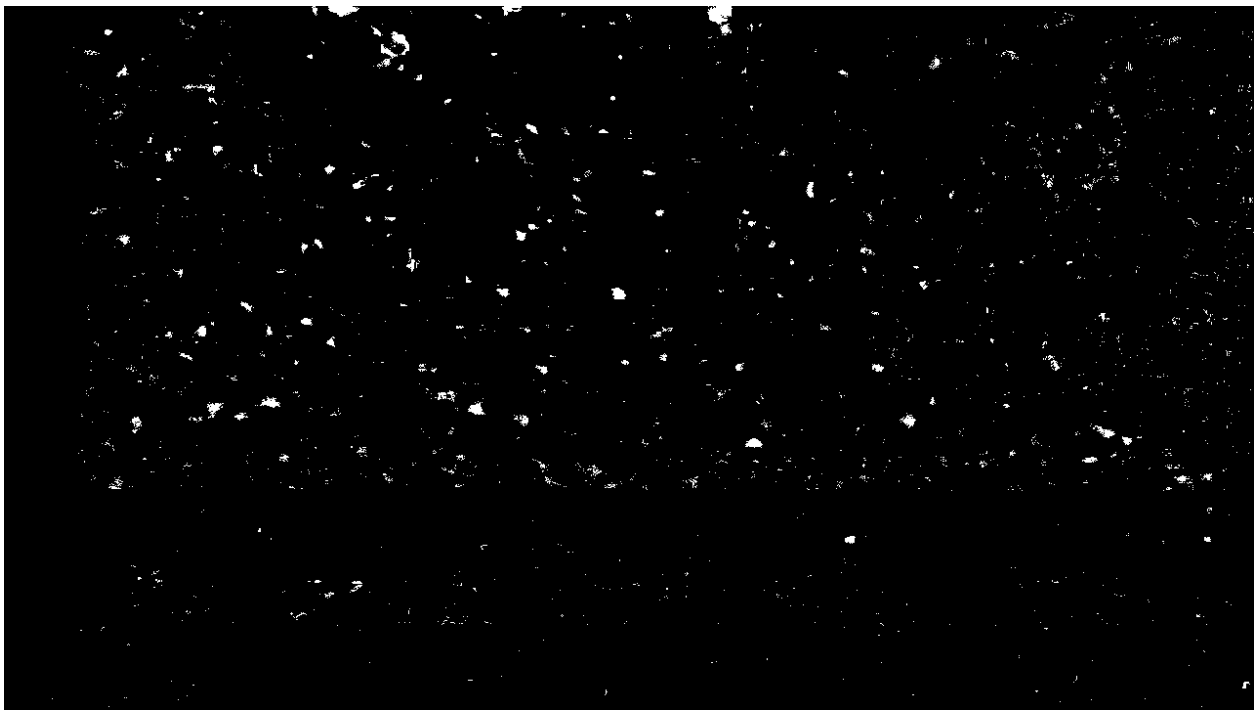


Figure 4.2: Welch's Test performed with alpha = 0.01 of the empty wing between the first and last 10 of 100 consecutively captured depth images.

BIBLIOGRAPHY

[1] S. R. Fletcher and T. L. Johnson, "Investigating the impact of product orientation on musculoskeletal risk in aerospace manufacturing," in *Contemporary Ergonomics and Human Factors 2014: Proceedings of the international conference on Ergonomics & Human Factors 2014, Southampton, UK, 7-10 April 2014*, CRC Press, 2014.

[2] B. Zhang, "Boeing is ramping up inspections after the us air force rejected its new kc-46 tanker planes again." https://www.businessinsider.com/boeing-ramp-up-inspections-after-us-air-force-rejects-kc-46-deliveries-2019-4, 2019. Online; Accessed 2-May-2019.

[3] J. E. Engel, R. S. Wright, C. H. Toh, and W. T. Edwards, "Methods and systems for enhancing backscatter x-ray foreign object debris detection," Sept. 24 2013. US Patent 8,542,876.

[4] J. Mund, A. Zouhar, L. Meyer, H. Fricke, and C. Rother, "Performance evaluation of lidar point clouds towards automated fod detection on airport aprons," in *Proceedings of the 5th International Conference on Application and Theory of Automation in Command and Control Systems*, pp. 85–94, ACM, 2015.

[5] D. Gates, "Boeing tanker jets grounded due to tools and debris left during manufacturing." https://www.seattletimes.com/business/boeing-aerospace/boeing-tanker-jets-grounded-due-to-tools-and-debris-left-during-manufacturing, 2019. Online; Accessed 2-May-2019.

[6] G. Legnani, F. Casolo, P. Righettini, and B. Zappa, "A homogeneous matrix approach to 3d kinematics and dynamicsi. theory," *Mechanism and machine theory*, vol. 31, no. 5, pp. 573–587, 1996.

[7] P. Owan, C. Devine, and W. T. Piaskowy, "Corerobotics: An object-oriented c++ library with cross-language wrappers for cross-platform robot control," *The Journal of Open Source Software*, vol. 3, no. 22, p. 489, 2018.

[8] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[9] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, (Shanghai, China), May 9-13 2011.

[10] D. Holz, A. E. Ichim, F. Tombari, R. B. Rusu, and S. Behnke, "Registration with the point cloud library: A modular framework for aligning in 3-d," *IEEE Robotics & Automation Magazine*, vol. 22, no. 4, pp. 110–124, 2015.

[11] A. W. Fitzgibbon, "Robust registration of 2d and 3d point sets," *Image and vision computing*, vol. 21, no. 13-14, pp. 1145–1153, 2003.

[12] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[13] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*, pp. 21–37, Springer, 2016.

[14] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 652–660, 2017.

[15] "Opencv/c++ connect nearby contours based on distance between them." https://dsp.stackexchange.com/questions/2564/opencv-c-connect-nearby-contours-based-on-distance-between-them, 2010. Online; Accessed 22-May-2019.

[16] "Draw bounding box around given size area contour." https://stackoverflow.com/questions/23398926/drawing-bounding-box-around-given-size-area-contour, 2014. Online; Accessed 22-May-2019.

Appendix A

ROBOT PROTOTYPE

Mechanical Design

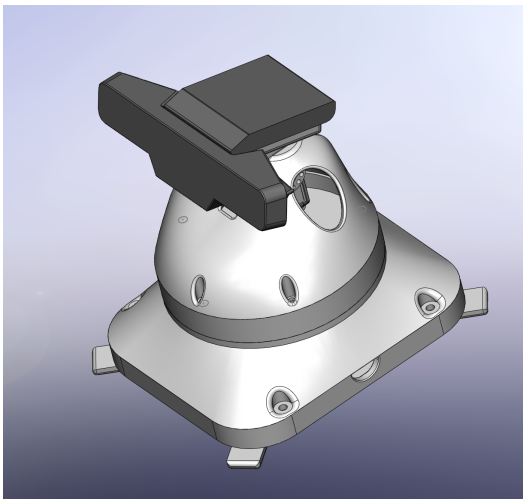


Figure A.1: SolidWorks CAD model

A SolidWorks CAD model for the RVA is shown in Figure A.1. The prototype features a 3D printed body, a captured first joint, and two degrees of freedom for pan and tilt control of a depth camera via two smart servos. The robot is small enough to fit through an access hole on a wing, and offers good visibility because of its servo-controlled joints.

Actuation

Two Herkulex DRS-0101 smart servos are used to control each of the robot's axes. Control commands are sent to an Arduino MEGA 2560 over USB from a host computer, and then translated into serial commands for the servos using the Herkulex library. Figure A.2 shows

one of the servos mounted in the robot chassis.

Figure A.2: Herkulex DRS-0101 smart servo mounted in the robot

Sensing

The Intel RealSense D435 depth camera is used. Note that the SR300 camera is shown in the CAD model (Figure A.1) but was not used for any results presented in this research. The depth camera provides both a stream of color images and a stream of depth images.

Software

Software for the robot is written primarily in C++ and Python, using CMake to automate Makefile generation and Make to compile the C++ executables. Point cloud processing is done primarily with the Point Cloud Library (PCL) [9]. Intel's open source library librealsense is used for gathering data from the depth camera, while Boost libraries are used for asynchronous serial communication with the Arduino. The CoreRobotics library is used for handling transformations between joints of the robot [7]. Reference commands are sent as a character string from the computer to the Arduino's USB serial buffer. The commands are parsed and sent to the servos using the Herkulex.h library.

Experimental Setup

For the RVA prototype, a tether is used to supply power and control commands from outside of the wing. The USB 3.0 cable for the D435 depth camera is run separately. An Intel NUC computer is mounted to an aluminum ground plate along with an Arduino MEGA 2560. 5V and 12V power supplies are also included to power the Arduino and servos, respectively.

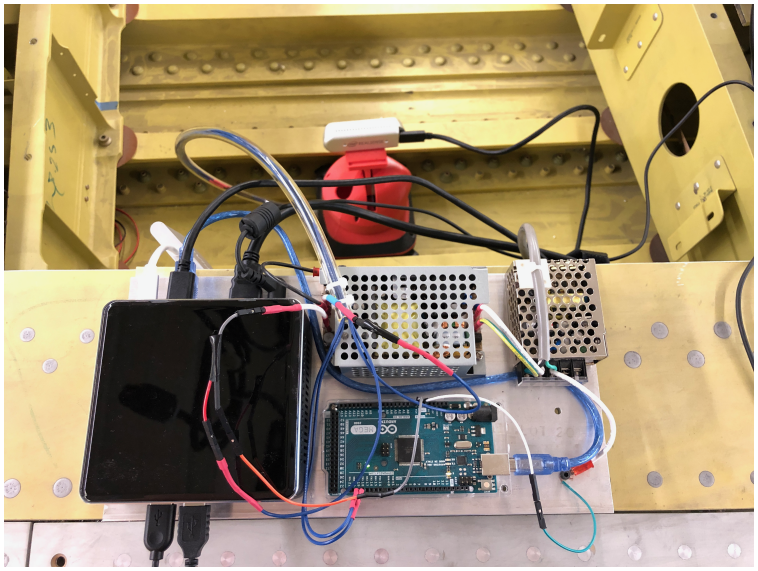


Figure A.3: Experimental setup

Appendix B

CODE

Detection Visualization

The function `drawChanges` is implemented in Python and based off of [15] and [16]. The algorithm takes a color image and binary image as inputs, and returns the color image with bounding boxes to the calling program.

```
def drawChanges(self,
                color,
                change_mask,
                filter_min_blob_size_pixels=0,
                filter_blob_max_consolidation_distance_pixels=30):

    # Remove small blobs from the binary image, which may be noise or irrelevant
    num_components, labels, stats, centroids = cv2.connectedComponentsWithStats(change_mask, connectivity=8)
    # Remove the background, which is also considered as a component
    sizes = stats[1:, -1]
    num_components -= 1
    diff_binary_filtered = np.zeros(change_mask.shape, dtype=np.uint8)
    for i in range(num_components):
        if sizes[i] >= filter_min_blob_size_pixels:
            diff_binary_filtered[labels == i + 1] = 255

    # Find Contours in the image, only supported for CV_8UC1 images
    contours, hierarchy = cv2.findContours(diff_binary_filtered, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
    contour_img = copy.copy(color)
    contour_color = (0, 255, 0)

    # Go through all the contours and combine contours that are close to each other
    def find_if_close(cnt1, cnt2):
        row1, row2 = cnt1.shape[0], cnt2.shape[0]
        for i in range(row1):
            for j in range(row2):
                dist = np.linalg.norm(cnt1[i] - cnt2[j])
                if abs(dist) < filter_blob_max_consolidation_distance_pixels:
                    return True
                elif i == row1 - 1 and j == row2 - 1:
                    return False

    LENGTH = len(contours)
    status = np.zeros((LENGTH, 1))

    for i, cnt1 in enumerate(contours):
        x = i
```

```
    if i != LENGTH - 1:
        for j, cnt2 in enumerate(contours[i + 1:]):
            x = x + 1
            dist = find_if_close(cnt1, cnt2)
            if dist == True:
                val = min(status[i], status[x])
                status[x] = status[i] = val
            else:
                if status[x] == status[i]:
                    status[x] = i + 1

unified = []
maximum = int(status.max()) + 1
for i in range(maximum):
    pos = np.where(status == i)[0]
    if pos.size != 0:
        cont = np.vstack(contours[i] for i in pos)
        hull = cv2.convexHull(cont)
        unified.append(hull)

# Draw a bounding box around each of the unified contour regions
res_depth = change_mask.shape
res_color = change_mask.shape
for contour in unified:
    rect = cv2.boundingRect(contour)
    x, y, w, h = rect
    x = (int)(x / res_depth[1] * res_color[1])
    w = (int)(w / res_depth[1] * res_color[1])
    y = (int)(y / res_depth[0] * res_color[0])
    h = (int)(h / res_depth[0] * res_color[0])
    cv2.rectangle(contour_img, (x, y), (x + w, y + h), contour_color, 2)

return contour_img
```

Servo Control

The following code can be flashed to an Arduino MEGA 2560 to control the servos, after setting baud rates and servo IDs appropriately.

```
#include <Herkulex.h>

String positionInput1 = "";
String positionInput2 = "";
float positionOutput1; // From position feedback
float positionOutput2;
String positionToSend;
String output;
boolean readyToWrite = false;
boolean delimiterRead;
boolean returnPosition;

// Servo IDs
const int servo1 = 1; // Servo ID
const int servo2 = 2;

// Soft limits
const int angle1Min = -91;
const int angle1Max = 91;
const int angle2Min = -91;
const int angle2Max = 10;

int pTime;
const int cutoff = 20;
const int vMax = 90; // Degrees/sec maximum servo speed

// for position feedback timing
long previousMillis = 0;
unsigned long currentMillis = 0;
long interval = 25; // milliseconds between position feedback send

void setup()
{
  Serial.begin(115200);     // Open serial communications
  delay(250);
  Herkulex.beginSerial1(115200); // Serial Port 1 for servos
  Herkulex.reboot(servo1); //reboot first servo
  Herkulex.reboot(servo2); //reboot second servo
  delay(500);
  Herkulex.initialize(); //initialize servos
  Herkulex.clearError(servo1);
  Herkulex.clearError(servo2);
  Herkulex.setLed(servo1, LED_PINK);
  Herkulex.setLed(servo2, LED_PINK);
}

void loop()
{
  // Get position feedback
  currentMillis = millis();
```

```
  if(currentMillis - previousMillis > interval) {
    previousMillis = currentMillis;
    positionOutput1 = Herkulex.getAngle(servo1);
    positionOutput2 = Herkulex.getAngle(servo2);
    Serial.println(String(positionOutput1) + 'x' + String(positionOutput2)) + '\n'; // added newline to end


  // Write to servos if flag is raised
  if(readyToWrite) {
      if((positionInput1.length() > 0) && (positionInput1.toInt() > angle1Min) &&
         (positionInput1.toInt() < angle1Max)) {
          float newPos1 = positionInput1.toFloat();
          pTime = getPTime(newPos1, positionOutput1, vMax);
          Herkulex.moveOneAngle(servo1, newPos1, pTime, 1);
      }
      if((positionInput2.length() > 0) && (positionInput2.toInt() > angle2Min) &&
         (positionInput2.toInt() < angle2Max)) {
          float newPos2 = positionInput2.toFloat();
          pTime = getPTime(newPos2, positionOutput2, vMax);
          Herkulex.moveOneAngle(servo2, newPos2, pTime, 1);
      }
      positionInput1 = "";
      positionInput2 = "";
      readyToWrite = false;
  }

  // Get serial input and parse the two commands delimited by a '.' such as 45.90
  // returns current position
  delimiterRead = false;
  while(Serial.available() > 0) {
       char inputChar = (char) Serial.read();
       if(inputChar == '.') {
          delimiterRead = true;
       } else {
          if(!delimiterRead) {
              positionInput1 += inputChar;
          } else {
              positionInput2 += inputChar;
          }
       }
       delay(1);
  }
  if (positionInput1.length() > 0 || positionInput2.length() > 0) {
       readyToWrite = true;
  }
}
```