5D NONDESTRUCTIVE EVALUATION: OBJECT RECONSTRUCTION TO TOOLPATH GENERATION

By

Ciaron Nathan Hamilton

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Electrical Engineering – Master of Science

2021

ABSTRACT

5D NONDESTRUCTIVE EVALUATION: OBJECT RECONSTRUCTION TO TOOLPATH GENERATION

By

Ciaron Nathan Hamilton

The focus of this thesis is to provide virtualization methods for a Cyber-Physical System (CPS) setup that interfaces physical Nondestructive Evaluation (NDE) scanning environments into virtual spaces through virtual-physical interfacing and path planning. In these environments, a probe used for NDE mounted as the end-effector of a robot arm will actuate and acquire data along the surface of a Material Under Test (MUT) within virtual and physical spaces. Such configurations are practical for applications that require damage analysis of certain geometrically complex parts, ranging from automobile to aerospace to military industries. The pipeline of the designed $5D$ actuation system starts by virtually reconstructing the physical MUT and its surrounding environment, generating a toolpath along the surface of the reconstructed MUT, conducting a physical scan along the toolpath which synchronizes the robot's end effector position with retrieved NDE data, and post processing the obtained data. Most of this thesis will focus on virtual topics, including reconstruction from stereo camera images and toolpath planning. Virtual mesh generation of the MUT and surrounding environment are found with stereo camera images, where methods for camera positioning, registration, filtering, and reconstruction are provided. Path planning around the MUT uses a customized path-planner, where a $2D$ grid of rays is generated where each ray intersection across the surface of the MUT's mesh provides the translation and rotation of waypoints for actuation. Experimental setups include both predefined meshes and reconstructed meshes found from several real carbon-fiber automobile components using an Intel RealSense D425i stereo camera, showing both the reconstruction and path planning results. A theoretical review is also included to discuss analytical prospects of the system. The final system is designed to be automated to minimize human interaction to conduct scans, with later reports planned to discuss the scanning and post processing prospects of the system.

ACKNOWLEDGEMENTS

I would like to thank my family, including my mother Mary Hamilton and my brother and sister-in-law Brendan and Emily Hamilton on the encouragement required in order for me to finish up this thesis. I would also like to thank my professor and mentor Yiming Deng, as well as Mahmoodul Haq, Lalita Udpa, and Satish Udpa, for the great opportunity to work with the Nondestructive Evaluation laboratory (NDEL) and facility research insude of the Composite Vehicle Research Center (CVRC) at Michigan State University. I would also specifically thank my lab mates Oleksii (Alex) Karpenko and Eric Tarkleson for help with the design and implementation of the system which was used for this thesis report. Finally, I would like to thank the Institute for Advanced Composites Manufacturing Innovation (IACMI) for the opportunity to jointly work with research and development with this thesis report.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

KEY TO ABBREVIATIONS

CAD Computer-Aided Design

CPS Cyber-Physical System

DH Denavit–Hartenberg

ECT Eddy Current Testing

ICP Iterative Closest Point

MUT Material Under Test

NDE Nondestructive Evaluation

SF Scalar Field

SOR Statistical Outlier Removal

SNR Signal Noise Ratio

RGB Red Green Blue

RMS Root Mean Square

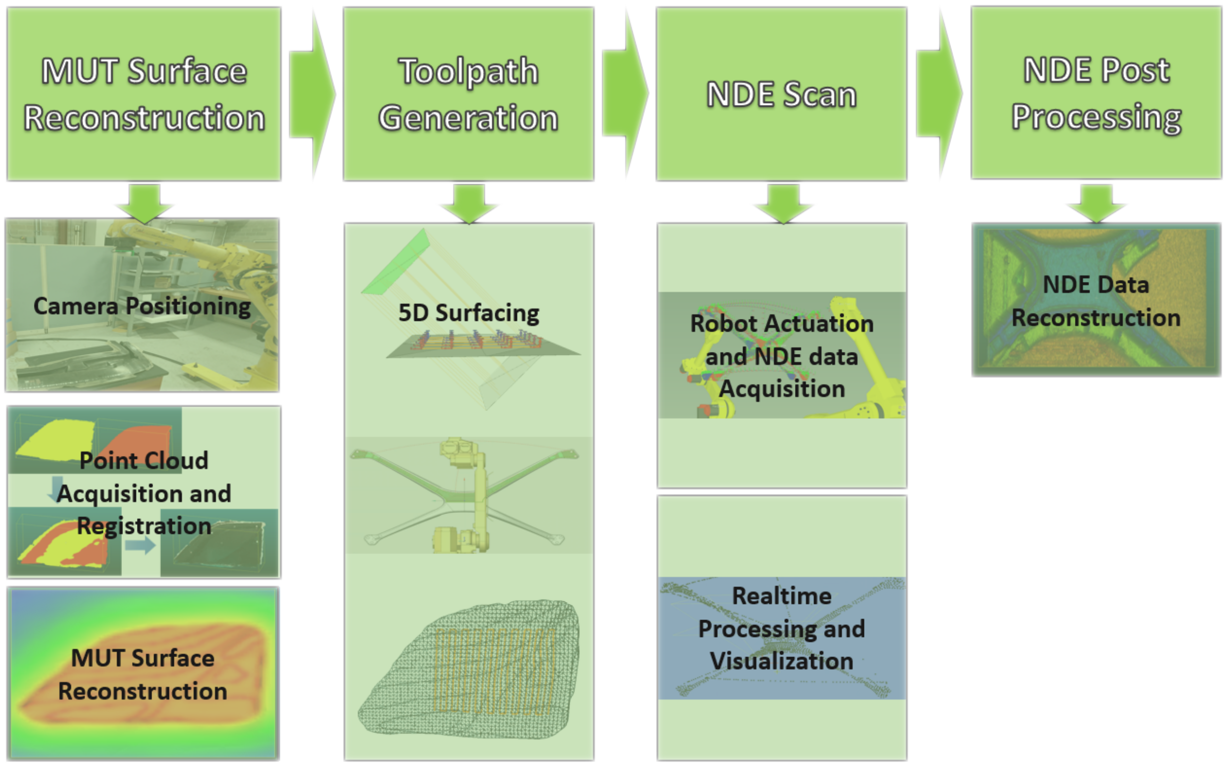
UT Ultrasonic Testing

TSP Traveling Salesmen Problem

CHAPTER 1

INTRODUCTION

Nondestructive Evaluation (NDE) is the examination of an object to parse certain descriptive material properties or conditions without damaging the entity itself [8]. One major study in this field is to determine the integrity of a Material Under Test (MUT) by finding irregularities which decrease serviceability. Within NDE studies of material properties, MUTs are generally flat panels in order to simplify the setup overall in order to focus on sensor interaction with materials to parse anomaly position and severity. Flat MUTs allow for the significant removal of variance in depth outside of a surface defect, simplifying the results for post processing and analytical models. These sensors could be used later from either a manual setup, where an inspector uses a hand device with the sensor over a certain MUT, or automated systems, in which actuation systems are used in order to obtain images or data-sets that are as outputs from the sensor. One automated way of obtaining NDE information from flat MUTs is to use an actuation device, such as a two or three dimensional gantry with orthogonal linear actuators, to run a $2D$ raster scan over the MUT at a constant lift-off distance between the probe and MUT's surface. This is a common method for testing qualitative results of the NDE sensor, with sensor position either closely approximated or known, while obtaining quantitative results through automation. The output would be a $2D$ data array that, possibly through some post processing, would show objective evidence of irregularities within the MUT with it's location relative to the scan being known. While from a research perspective this allows for more luxury to focus on material properties and interactions between the sensor and MUT, practical applications of NDE requires the determination of material integrity with objects of higher geometrical complexity. For automated NDE inspection systems, there are also considerations for mapping defects within a $3D$ environment, in which requires more complexity over $2D$ systems. The issue with $2D$ scanning is that there are many presumptions for sensor interaction with complex shapes that might effect how sensors operate within those environments. Factors such as effectiveness when a sensor is against curved surfaces, where it may be difficult to



MUT Surface Reconstruction
Toolpath Generation
NDE Scan
NDE Post Processing
Camera Positioning
5D Surfacing
Robot Actuation and NDE data Acquisition
NDE Data Reconstruction
Point Cloud Acquisition and Registration
Realtime Processing and Visualization
MUT Surface Reconstruction

Figure 1.1: 5D nondestructive evaluation cyber-physical system pipeline

keep the sensor completely normal to the surface in a practical setting, is not considered but may potentially render impracticality with the sensor itself, depending on gain requirements. Various fields, from automotive and aerospace industries to military applications, require regular diagnostics on critical system units to avoid cataclysmic or otherwise costly events. Having the best possible sensor with as close physical simulation as practical NDE applications is important, but practical components are generally non-flat or even geometrically complex. For example hydro turbines are susceptible towards multiple damage types, such as cavitation, erosion, material defect, and fatigue [6]. On a turbine failure, there will be an increase in power plant down-time and life-threatening hazards for operational and maintenance personnel. Aircraft components of various shapes are also susceptible to corrosion, which the failure to detect such anomalies is to blame for certain airline accidents in the past [3]. These are just a few examples among many. There is a need for automated examination methods increases to allow for high quality inspection while decreasing losses. This report discusses the virtual implementation of such a system with the usage of robotic arms that would act as a universal system for actuating over most materials under test within it's workspace. Such information of sensors inside controlled environments with geometrically complex objects is useful for determining the effectiveness of sensors within practical fields.

To allow for scanning of geometrically complex objects, an expansion upon the complexity of the NDE actuation system can be used in order to match the geometrically complexity of a MUT within an automated manor. In other words, the probe as the actuation system's end-effector must be able to position itself within regions that provide critical NDE data. Considerations would be samples too large for the workspace, complex concave properties, or regions where the actuating device itself would collide with the sample. The scanning system will require the same NDE principals learned from simpler actuation setups, such as obtaining a meaningful signal information from the probe with minimal Signal Noise Ratio (SNR) while maintaining near constant lift-off to reduce variant data based on lift off changes. Geometrical knowledge of the MUT, the probe and actuation systems, and other environmental objects such as the ground or sample holders are required to be known within virtual space in order to conduct scans without collisions. For noncomplicated setups, such as $2D$ or $3D$ gantry systems, by conducting metric measurements inside the actuator/gantry space to describe the differences between physical and virtual setups. Within a more complicated actuation setup, this comfort is removed and the reliance on Cyber-Physical System (CPS) and virtual-physical technologies are needed as a baseline to interface the physical and virtual testing environments including to properly conduct a valid and informative scan. The proposed actuation system pipeline is included within Figure 1.1, which shows the synchronous processes to take place in order to conduct an NDE scan within the extensions of space. This system will use an industrial grade Fanuc 100iB robotic arm which has the capabilities to reach within 1.37 meters withing a spherical actuation space. For the purposes of this report, only the virtual implementation of this robot will be used. The system is still in development, as there are various improvements on virtual-physical interfacing that need further validation as will be discussed in the conclusion of the report. In short, automated capabilities of the robot actuation were not used or examined for this report. The pipeline remains the same despite this complication.

The pipeline shown in figure 1.1 is implemented as an input/output system from left to right on the diagram. The first step is to obtain a Computer-Aided Design (CAD) reconstruction of the MUT that ensures the virtual-physical interface between the MUT and model. The resultant of this phase is then placed into a toolpath generation process, where waypoints of the sensor are defined and a potentially optimal path is solved to reach most or all points. The NDE scan phase will then use the generated path to conduct a scan, both virtually within a simulated phase and physically in a real test environment. For the sensor to reach all desired points, the position of the end-effector is found by using inverse kinematics calculations to solve robot arm joint solutions from a defined point within the boundaries of the environment. While a physical scan is conducted, the system should continuously obtain sensor pose and NDE data. The pose data should contain transformation information such as translation and rotation. Finally, the output data from the scan is interpreted and reconstructed via NDE post acquisition processing. Such would be to view the obtained data within a $3D$ CAD environment, with data either set as a colored point cloud with a color scale based on previously obtained data points, or as a reconstructed mesh when the scan is finished. Most work done for this report was due to integrating software holding certain important functions, such as for point cloud manipulation and reconstruction methods or inverse kinematics calculations, or creating custom software such as a path-planner for generating waypoints and a path along the normal of an input meshes surface. Both of these were required for an operable CPS, but still require some more work done i in order to have a fully automated system. As such, only object reconstruction and toolpath generation are discussed in this report. Also due to validation concerns, reconstructed models are assumed to be within the robotic environment but pose not fully aligned. Later reports should address this issue, as well as physical NDE scan results and post processing. This report discusses relevant theoretical aspects of the report in high detail, where separate user-friendly and programmer-friendly documentation is later developed for those who should later use or develop for this system.

CHAPTER 2

THEORY OF PHYSICAL AND VIRTUAL SYSTEM INTERFACING

To understand the expansion of actuation complexity, a breakdown of basic components and functionalities of a probe as the end-effector is required. Transformation vectors play a large role in the positioning of virtual components such as the MUTs mesh and robot arm kinematics. The transformation vectors mentioned will be used in regards to Cartesian space xyz, where translations and rotations occur across each axis. Note that scaling vectors for transformations are completely ignored due to irrelevance. The simplest unit of data is considered as a point, which will hold the translation vector $T_{xyz} = [T_x, T_y, T_z]$ and rotational vector $R_{xyz} = [R_x, R_y, R_z]$. Three types of points will be discussed: "CADpoints" used alongside CAD meshes, waypoints where an end-effector should reach, and "datapoints" which holds sensor data and lay along the path in which waypoints set up. The set of CADpoints used along meshes are called a point cloud. Using three or more of these points inside of a point cloud, a face is generated in which a normal vector can be parsed where $N_{xyz} = [N_x, N_y, N_z]$ and $|N_{xyz}| = 1$. Normal vectors only occur along one side of each face, in which direction is parsed based on the index position of each point. CAD points and faces are used in the reconstruction process where a point cloud is converted into a mesh, the path planning process which parses position and rotation from defined meshes, and the simulated NDE scan process for collision detection. A waypoint is inside a set of called the toolpath, and eventually are organized into a path vector, with a defined start and end point. Datapoints are organized alongside this path vector, but are otherwise independent as they are acquired through the scanning process rather than defined in the path planning process.

All types of points need to be considered as bounded within a workspace environment. In terms of continuous spatial parameters, the total boundary set or configuration space of the environment within actuation will be regarded as E. Within a Cartesian space, the boundary set can be considered as a relation between E and various E_{min} and E_{max} parameters, where E_{min} is the set of minimum values along translation $Txyz$ or rotation $Rxyz$, and likewise E_{max} is the set of maximum values along xyz. There may be several combinations of elements between E_{min} and E_{max} parameters to describe complex spaces for each T_{xyz} and R_{xyz}, therefore this definition is more suitable for simple systems. For example, a $3D$ gantry without a rotational probe on it's end-effector is bounded by it's minimum and maximum T_{xyz} values within a cuboid based directly on the length of the linear actuators running the system, and the end-effector presumably pointing downwards against the earth with $R_{xyz} = [0, 180°, 0]$ or $N_{xyz} = [0, 0, -1]$. A robot arm is another example where the space is defined within a spherical region based on the possible reaches of the robot and rotational space defined by it's Denavit–Hartenberg (DH) displacements for each kinematic chain. This example requires more complicated definitions when considering a robot arm system due it's sphere-like workspace. Such would require a large number of definitions depending on combinations between translations and rotations along xyz axis. Therefore, it might be better to describe E with respect to robot joints, with J being equivalent to the set of all robot joints, J_min being the total set of maximum rotation of a joint at a certain index, and likewise J_max being the minimum rotations of a joint. Through inverse kinematics of DH parameters, transformation parameters can be parsed. Joint space descriptions are useful for accurately describing robot configurations, however are less comprehensible for approximates location of joint components and as such the end-effector due to various changes in reference frames based on arm lengths, requiring some computation before giving an understanding of it's location. As such, Cartesian descriptions are generally shown instead of joint configurations, and boundaries are approximated.

When data is taken from a sensor, a discrete number of waypoints and datapoints will be acquired along which is determined from the probes position with respect to the environments space and the data acquired from the sensor. The total set of waypoints in the toolpath are considered is defined as $S = [s_0, s_1, ..., s_n]$ with $s_k = [T_{xyz}, R_{xyz}]$ where s_k is an indexed point with $k = 0, 1, ..., n$ and path size n. Ideally, the rotation of each point should be along the surface but other configurations are possible i.e. constant rotation scans. Normals are not considered if the start and stop waypoints are away from the MUT i.e. a known home position. The robot should actuate towards each point in S, where the end-effector transforms at any given time is considered as d and the total

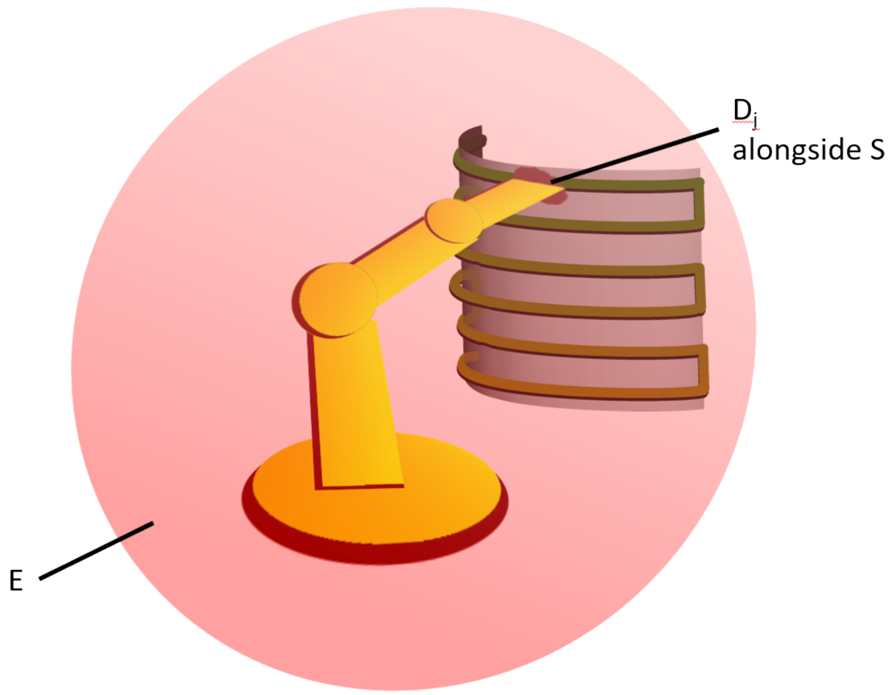


Figure 2.1: Robot arm NDE scanning environment along a curved sample

list being D. Indices inside D are independent of S due generation through different processes, so d_j is an indexed point where $j = 0, 1, ..., m$ where m is the number of acquisitions throughout the scan. If a mode such as point-by-point is used, where each acquisition is done at each way point, then $n = m$ and approximately $S = D$, though there may be noise between physical location. Otherwise, then any d_j in D will lay approximately on a theoretical continuous path S_{cont}. An basic scanning setup is shown in figure 2.1, showing the environment E with spherical bounds by the robot arm, the continuous path S_{cont} on a curved sample in which a discrete point d is taken. Toolpaths for scanning may contain different possible configurations, but such paths for this report will considered as raster-like, where the end-effector will move in a zig-zag pattern along the desired region of interest. This "region of interest" can be the entirety of the sample, or a zone within the sample considered as a segmented scan. Usage of segmented scans may be advantageous when only certain regions should be examined within the NDE scan, and may simplify the scanning process in turn shortening the amount of time to conduct an NDE scan. Multiple segmented scans can be partitioned to examine larger regions of interest.

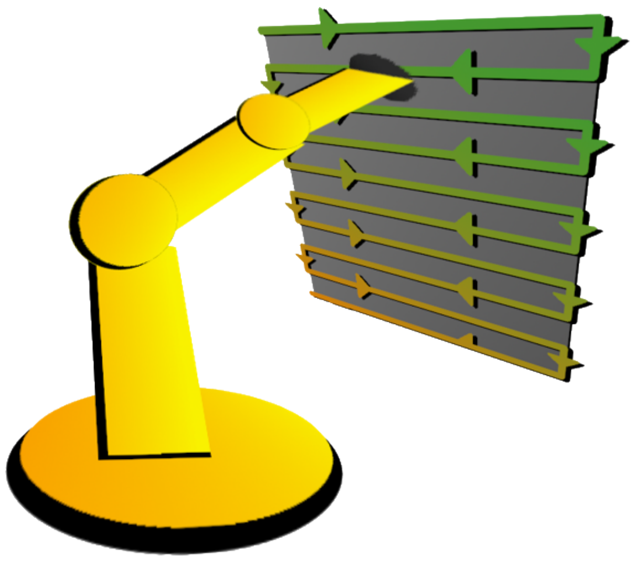


Figure 2.2: 2D flat panel scan

There are several considerations for different shapes and configurations of MUTs. For example, figure 2.2 shows a $2D$ flat panel scan using a $3D$ gantry assuming alignment on the z-axis. Each point in the toolpath will contain $p_k = [T_{xyz}, R_{xyz}] = [< T_x, T_y, c >, < 0, 180°, 0 >]$, where T_x and T_y are move variably alongside the flat sample and $T_z = c$ is a constant lift-off, the probe faces downwards against the MUT where $N_{xyz} = [0, 0, -1]$, or $R_x = R_z = 0$ and $R_y = 180°$. For extension into $5D$ motions, each movement acknowledges all translation parameters T_{xyz} and two rotational R_{xy} with respect to the end-effector to mimic spherical coordinate rotations within Cartesian space. For rotational coordinates, R_{xyz} is used despite this simplification as this definition is used throughout software implemented for this report.

Within possible movements, the boundaries of the environment itself are not the only concern. If there are any obstacles inside of the environment, such as the floor, surrounding walls, or apparatus holding the MUT, they must be avoided. Consider any obstacle region as O and toolpath S_{cont}, then $S_{cont} \cap O = \emptyset$. As such, $D \cap O = \emptyset$. This behavior is a simplification of the collision detection process, which include body comparisons of objects within virtual space, however as a basis it is required that any points in the toolpath do not conflict with any obstacles. The MUT itself should also be considered as an obstacle to avoid collisions with, but the system will also need to know where the MUT is to scan in particular to remove lift-off or rotational errors. Hence why the reconstruction of a virtual system containing body elements inside the environment is highly suggested for $5D$ setups. It is possible to contain physical measurements setup for the same results, but such tactic will lead to laborious and less robust setups.

Upon the measurement of S data will also require NDE data at the same point, which depending on the type of NDE being conducted. These points holding both types of information are considered at "NDEpoints". The set of NDEpoints is D_{nde} and an individual NDEpoint is d_{nde_j} for $j = 0, 1, ...m$. Through inheritance, D_{nde} is a subclass of D with d_{nde} being a subclass of d. D_{nde} holds the information to store NDE data and visualization properties and it's transformation parameters within virtual space. NDEpoints will include data such as but definitely not limited to Eddy Current (ECT) or Ultrasonic Testing (UT). Some NDE methods applied may have an abundance of different data types obtained real-time. In particular with UT, an array of data with respect to time is obtained per point which can be processed to determine the time-of-flight, energy, and amplitude within a single value each. NDE data will be acknowledged and used for further post processing purposes, and this information is considered as a block of data per point, or $Block_{nde}$. Finally, for human visualization purposes, single NDE values may be colorized using a Red Green Blue (RGB) color vector C_{rgb} using a normalization map N_{map} of possible minimum and maximum values. In summary of what an NDEpoint will include are the transformation parameters T_{xyz} and R_{xyz}, the NDE data block which will be different based on the field of NDE used, and the color vector C_{rgb} based singular NDE values parsed from the NDE data block, hence the definition of the each NDEpoint as $d_{nde} = [d, Block_{nde}, C_{rgb}]$.

An NDEimage, I_{nde}, is formed as a cluster of NDEpoints generated from the scan, where $D_{nde} \in I_{nde}$. The NDEimage I_{nde} will also include the normalization map, so $N_{map} \in I_{nde}$. The entire set of NDE possibilities in the system is regarded as X_{nde} where $I_{nde} \subset X_{nde}$. Figure 2.3 shows the hierarchy of all the contents of $Inde$. $Block_{nde}$ will contain one NDE data set or multiple, or fused data or anything else, hence why "NDE Method x" is mentioned as a wildcard sort of NDE block type as many possible methods may be used as an input. I_{nde} will only contain a very

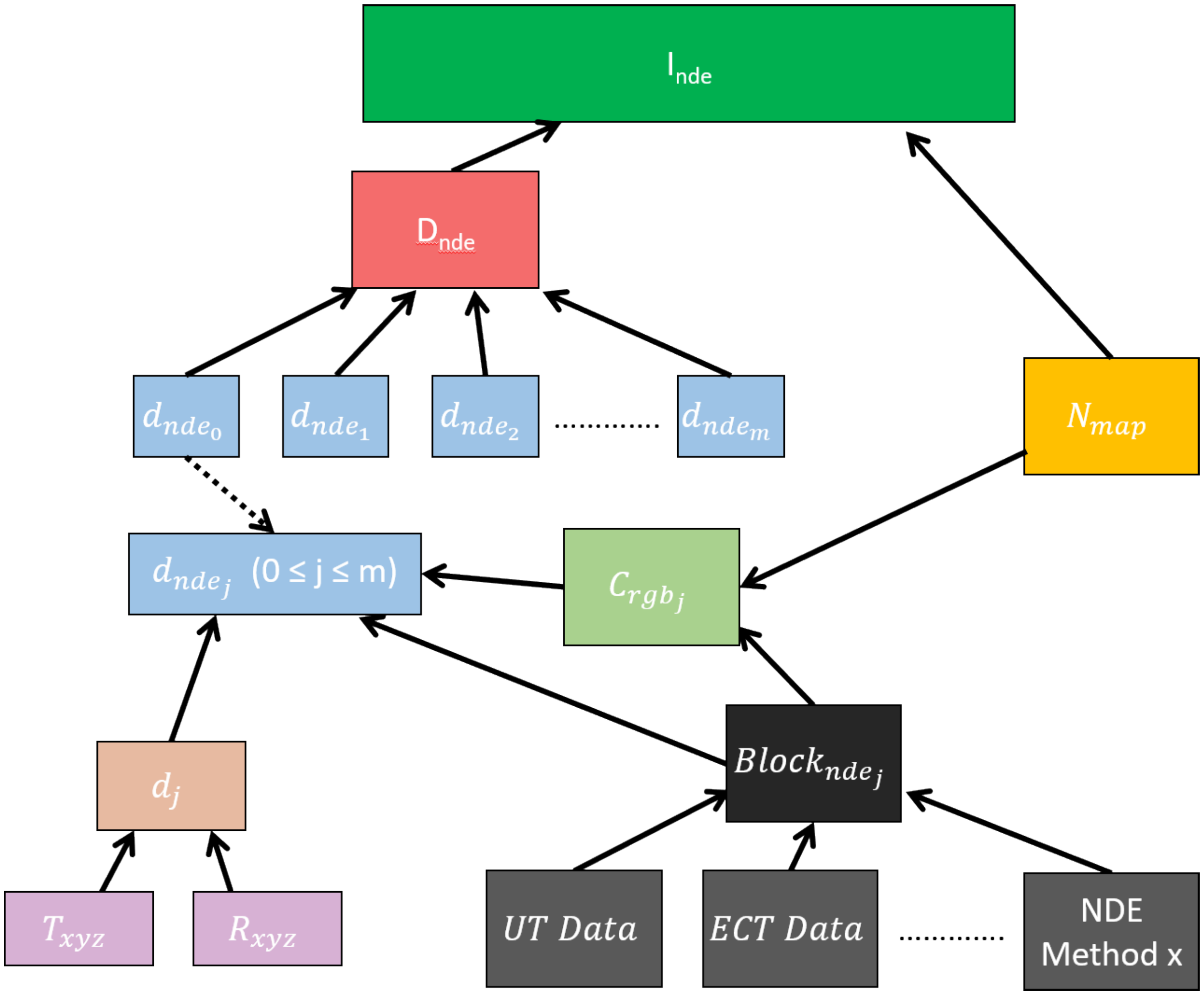


Figure 2.3: NDEimage hierarchy

small subsection of data, technically infinitesimal as X_{nde} is continuous, but is defined this way incorporate the multiple possibilities within a scan which should be avoided. In the broadest sense, then $X_{nde} \subset X$, where X is the set of entire physical phenomena within the environment not just included via information restrictions based on NDE.

In terms of visualization of NDEimages, there is an increased amount of complication for how to view such images between $2D$ and $5D$ setups. In a $2D$ flat panel setup, the compilation if NDEpoints within the NDEimage is generally uniformly distributed as "2D pixels" for the simplicity of visualization and formatting into a $2D$ image. Even in cases where the acquisition of points is not synchronized with these pixels, such as in cases where NDE data is desired to be obtained as fast as possible, generally interpolation methods with static placed pixel locations are used. In $5D$ setups, then it becomes increasingly difficult to normalize into a $2D$ image. Therefore, the advancement into virtualization technology is required, where each NDEpoint will be placed into a $3D$ point cloud in which point inside holds d and C_{rgb}. Reconstruction of the the point cloud into mesh possible, where normalization vectors are important for determining alignments along the mesh. Various filters can be used to colorize the point cloud based on depth or normal positions as well, or to remove noise or smooth meshes as explained later. Mesh reconstruction and filtering are processes that can be used for both the mesh into the path planner and for I_nde for data post processing. A framework for the pipeline to be explained within the next chapters is then set forth. The surface MUT reconstruction is placed so that a bounded path based on which points are required to scan but not including paths conflicting with obstacles and the MUT itself. The toolpath generator will use the resulting mesh to generate the points used for an NDE scan, which will obtain an NDEimage used later for post processing purposes.

CHAPTER 3

MUT SURFACE RECONSTRUCTION

Obtaining the CAD model of the MUT is required for the generation of toolpath used for the robot actuation. Finding this CAD using stereo camera and point cloud technology allows for robust scan setups and physical measurements may be tedious and lead to limited approximations for the scan. The lower noise within the reconstruction will mean a closer estimation where the probe to be placed alongside the MUT Within this setup, no laborious measurements to ensure the MUT is aligned with the robots are required, and that collisions between the probe end-effectors and a misaligned MUT can be prevented. A predefined CAD model for the MUT generated from a CAD-based software is also an option useful for simplifying the process and avoiding reconstruction overhead, but such requires accurate physical measurements of the setup since relative positioning can become complicated. It is possible that this step may be skipped for systems where the MUT and environment is completely known and not required for automation. It is also possible to avoid steps such as registration if a single stereo image can describe the region used for the surface scan. Other surface acquisition methods separate from using a stereo camera may yield better results, such run preliminary scan with a depth sensor to obtain a point cloud, where only one point is found at a time rather than an entire image including depths. These methods may also be more time consuming. For the sake of this report, only stereo camera methods have been examined as a physical system to actuate a depth sensor would require more development.

There are three main steps required for proper reconstruction to be placed. First is the camera positioning, where a singular image or combination of images of the environment are to be obtained. Next is the acquisition and registration where the stereo camera image is obtained, filters are applied, and adjusted to match multiple images together within the virtual space. Finally is the CAD reconstruction where the point cloud is converted to a $3D$ model with defined normal values per point. This reconstructed value will then be used for toolpath planning.

3.1 Camera Positioning

The first step of the CAD reconstruction process determining where to position the camera in order to obtain quality information. Consider the stereo camera to as any ordinary camera. One single shot might give a lot of detail about the scene and objects within the scene, however, there will be missing information for anything behind any object within the scene. Consider this missing information to be a "shadow" which is cast away from the camera. To counter this, it would be possible to move the camera into a different position to obtain more information behind objects which obstructed the first image, giving more information about the object and the scene itself. This idea extends toward usage of stereo cameras, where the surface details of the image are provided within a point cloud based measured depth measurements and binocular intrinsic parameters. Multiple surface images as point clouds can be amassed from different viewing points, which aligned or registered together provide $3D$ data to be fully reconstructed. Positioning requirements would mostly require on the experimental setup, such as if a single sided scan is available versus a dual sided set up, or the geometric complexity of the MUT.

Figure 3.1 shows a rough idea of a single camera setup against a MUT. Assuming the depth of the image is uniform so the distance between the camera and the MUT is d directly alongside the z-axis. The total set of elements in the point cloud is P_{Cam} which contains points $p_{Cam_{ij}}$, where $0 \leq i \leq w$ and $0 \leq j \leq h$. As explained in the theory section, a point p contains the positional and rotational values of the point within space. P_{Cam} is bounded by the resolution of the camera, defined by width w and height h. There is also a C_{rgb} value that may be provided if the camera has RGB capabilities. Therefore any point within P_{Cam} is defined as $p_{Cam_{ij}} = [p_{Cam_{ij}}, C_{Cam_{ij_{rgb}}}]$. The point cloud also defined as a body, where all points may be transformed alongside a center point.

One key component required for registration is knowing the transformation parameters Cam_{xyz} of the stereo camera once an image is shot, with respect to the scanning environment. This allows for vast simplification of post processing as the point cloud can be simply translated and rotated based on these parameters. A possible setup would be multiple cameras placed in static positions, where the Cam_{xyz} is always known and number of camera's and positions can be scaled based on the

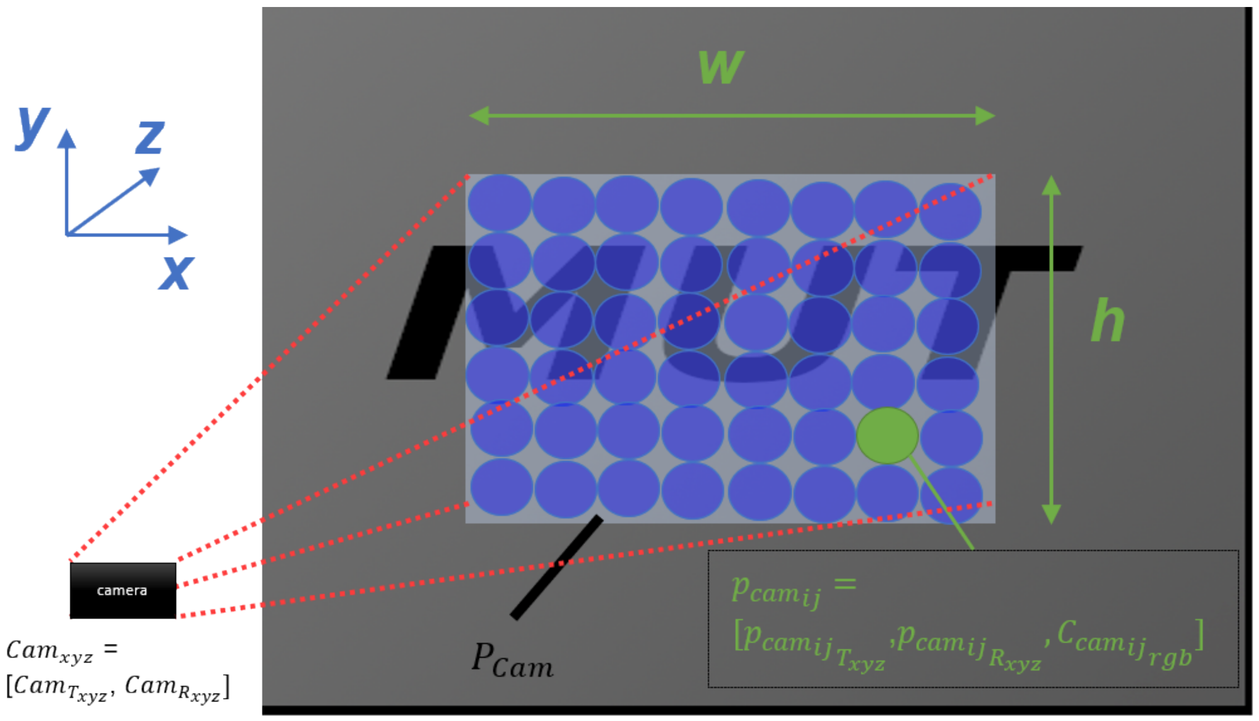
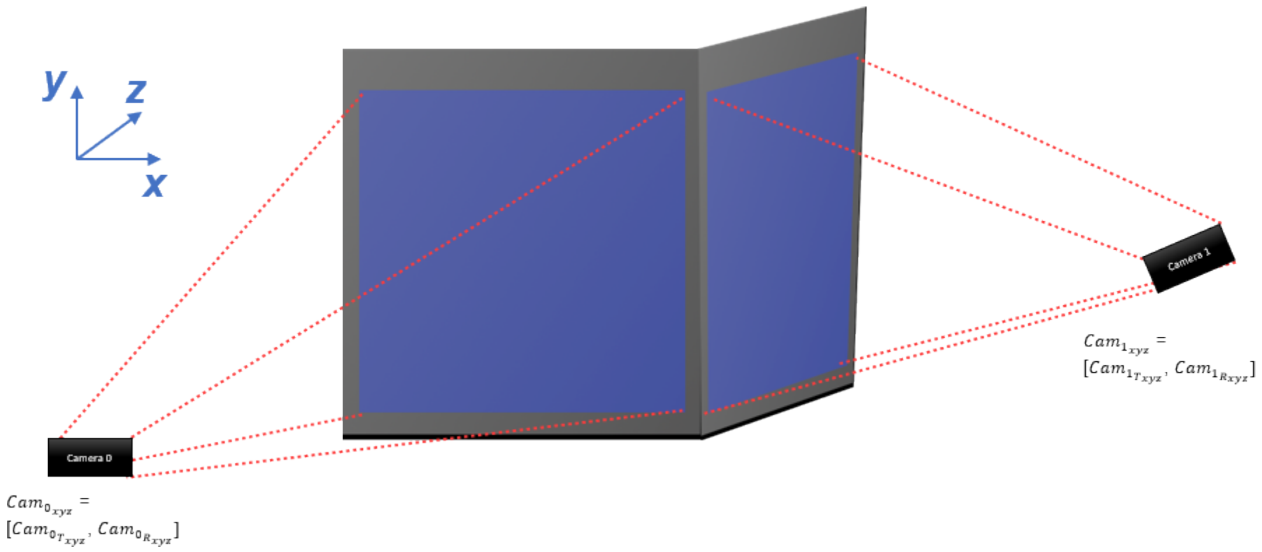

y
z
x
w
h
camera
$Cam_{xyz} = [Cam_{T_{xyz}}, Cam_{R_{xyz}}]$
P_{Cam}
$p_{cam_{ij}} = [p_{cam_{ij_{T_{xyz}}}}, p_{cam_{ij_{R_{xyz}}}}, c_{cam_{ij_{rgb}}}]$

Figure 3.1: Single camera position against a flat panel

MUT scanning requirements. Multiple camera positions are defined by $Cam_{k_{xyz}}$ with $0 \leq k \leq n$ with n being the number of cameras. This configuration is shown in figure 3.2. Since the cameras and MUT are static in this setup, then the parameters for registration via center point transformation are always known. A second suggested setup would be to have the stereo camera itself as an end-effector of an actuation system, where the Cam_{xyz} is known from the forward kinematics of the actuation system. This setup is shown in figure 3.3 for a single robot, and in figure 3.4 using a dual robot setup. In the hemispheroid setup, each robot scans a hemisphere mirrored from each other, creating a spherical path around the sample to obtain the full geometric capture of the MUT, with the resolution based on the number of shots and the depth resolution of the camera itself. The advantages of applying this setup over a multiple camera setup is that it removes the requirement of placing several cameras, however it requires the workspace dimensions and valid positions to be suitable for the reconstruction. The definition of multiple camera snapshots similar to the multiple static camera setup, where $Cam_{k_{xyz}} = d + Cam_{padding}$ with f as the position of the robot's end-effector, $Cam_{padding}$ is the physical size of the camera that would shift the position in virtual space slightly, and $0 \leq k \leq n$ with n being the number of shots the camera has taken defined by the path the robot took in the process of reconstructing the MUT. For simplification, $Cam_{padding} = 0$ and $Cam_{k_{xyz}} = d$, however knowing $Cam_{padding}$ may be a significant factor in minimizing registration errors. The number of shots is determined by how many movements the actuator can make with time, and is bounded by the actuation system's workspace. The camera itself also has minimum and maximum range requirements with respect to the physical surface to reconstruct. This is a consideration, assuming the robot used for NDE scanning also holds the camera for reconstruction.



y
z
x
Camera 0
$Cam_{0_{xyz}} = [Cam_{0_{T_{xyz}}}, Cam_{0_{R_{xyz}}}]$
Camera 1
$Cam_{1_{xyz}} = [Cam_{1_{T_{xyz}}}, Cam_{1_{R_{xyz}}}]$

Figure 3.2: Multiple camera setup

3.2 Point Cloud Acquisition and Registration

Once the stereo camera is properly positioned, either single or multiple point clouds may be obtained. The point cloud $Cloud_{xyz}$ will contain a discrete number of points which are normalized between a set amount of x and y pixels with various depths z. Several filtering methods are used on image acquisition as provided by Intel RealSense's C++ SDK [1] [2] in order to decrease various noises background data that may conflict with surface reconstruction later. In order of usage, first is the decimation filter which reduces the complexity of the image which saves time in path planning processing data later. Later decimation filters can be used to reduce complexity of registered images. The decimation filter uses kernels of size $2x2$ and $8x8$ along size the width and height

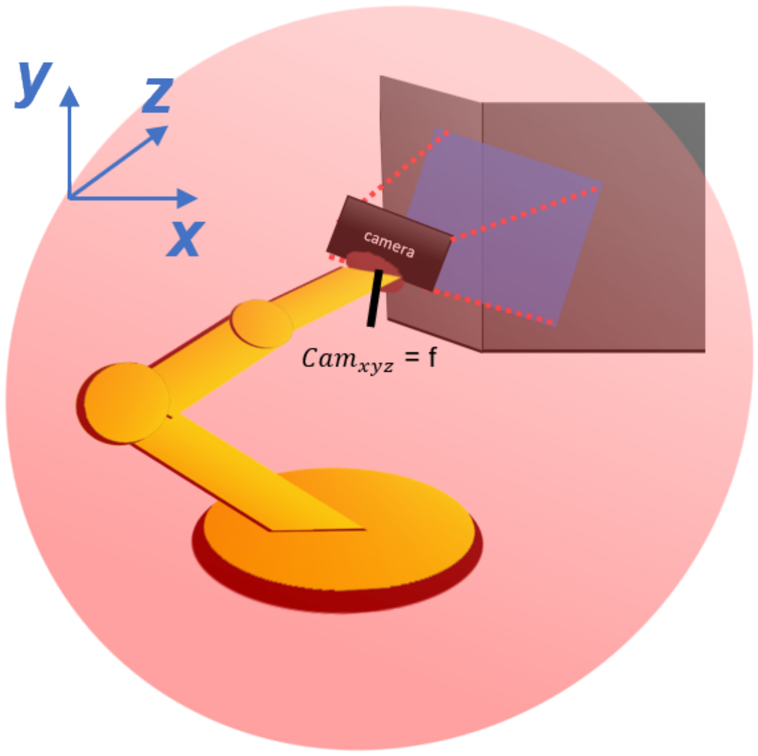


Figure 3.3: Camera as end-effector setup with a robot arm

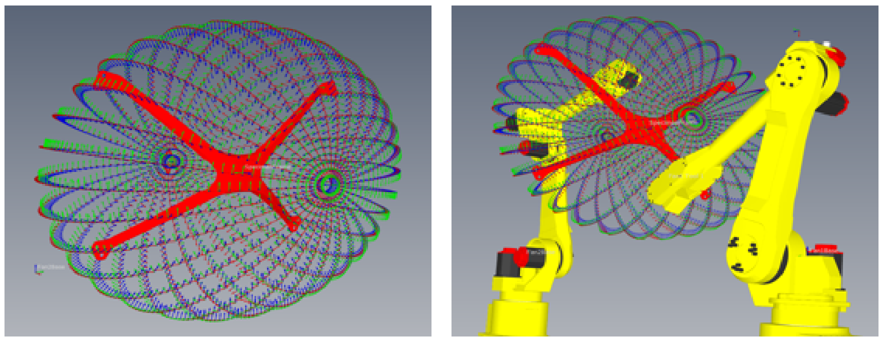


Figure 3.4: Hemispheroid scan with dual robots within RoboDK

of the image, adjusting the number of x and y pixels in the process. This filter simply calculates the median depth value of those selected points. Depending on the computation requirements and number of images obtained should determine the heavier usage of this filter. The second filter is a threshold filter, which a foreground and background value is selected to zoom in on the regions of interest, in turn removing a significant amount of background information. Next is the spatial edge-preserving filter which smooths out the data by using various 1D horizontal and vertical iterations [7]. The temporal filter is the forth in series, which uses previous point cloud images to improve persistent information. Due to the nature of preserving previous data, the temporal filter is best fist for static images and therefore has limited usage if the camera is used as an end-effector of an actuation system, unless there are pauses to add time for the filter to adjust. In short, for the best usage of this filter, the camera should be set steady. The final filter is the hole filling filter, which tries to find four neighboring pixels to remove any holes within the points to reconstruct.

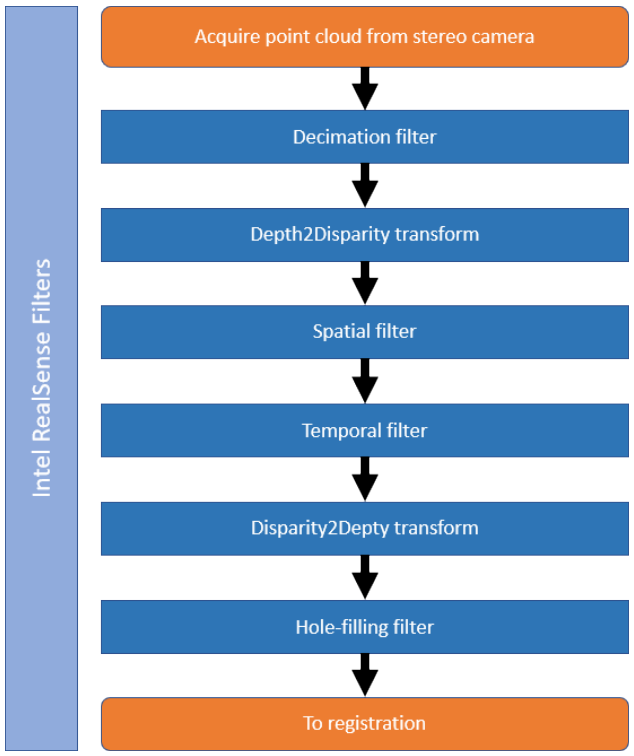


Figure 3.5: Intel RealSense filters applied on point cloud acquisition [2]

Registration of the image or bundle of images is required to align the virtual output of the stereo camera into physical space. If multiple images from various angles were obtained, then registration methods are used to reposition the point clouds so they create a surrounding image for the $3D$ reconstruction of the MUT. One method explored was center point transformation, where a position vertex is selected within the point cloud in which the entire point cloud is rotated then translated. Center point transforming requires both the translational and rotational properties of the camera with respect to the scanning workspace. Therefore, the output point cloud transformation at Cam_{xyz} with elements $Cam_{k_{T_{xyz}}}$ and $Cam_{k_{R_{xyz}}}$ as matrices allows for the equation $Cam_{xyz} = \sum_{k=0}^{n}(Cam_{k_{R_{xyz}}} * Cam_{k_{T_{xyz}}})$. The functionality of the summation operator here is to append the points together into Cam_{xyz}. The transforming process is automatically done through MeshLab scripting in Python. All point cloud images are merged into one image after registration.

Figure 3.6 shows a center point transformation with a point cloud MUT model generated from a predefined curved surface. This example is extendable towards usage of registering multiple point clouds, assuming proper Cam_{xyz} parameters for each shot taken. The "dots" in front of the point cloud and the CAD indicate the location of the center point of the models. This would be where there camera obtained the point cloud shot within virtual space, assuming a decent distance between itself and the MUT. Rotations occur around the center point, while translations move both the center point and the point cloud model. In the example, the MUT model is defined to be away from the destination at $MUT_{T_{xyz}} = [-1, 3, 0]m$ and $MUT_{R_{xyz}} = [0, 0, 90]deg$. The center point of the MUT is $-5m$ away from the center of the point cloud center alongside it's relative x-axis. To register the point cloud to it's destination, the model must first be rotated 90 degrees along the z-axis, then translated $T_{xyz} = [1, 3, 0]m$. The order of these transforms mater due to matrix multiplication, therefore translation before rotation is not acceptable. After the transformation, then the MUT is aligned with the destination MUT and is considered registered. In practice, measurement errors of where the camera resides in physical space and the virtualization of the MUT will cause errors in alignment.

If the translational and rotational transformation parameters of the camera within the environment are not available, then it is possible to use other registration methods such as manual point pair selection or automatic fine registration via iterative closest point (ICP). Both use a reference mesh in which other meshes to align will be transformed to match the reference mesh based on certain geometric similarities between the meshes. Manual point pair selection was not explored in much detail due to it requiring manual user inputs. ICP was explored when using methods where the multiple point clouds would be misaligned, and is explored in the experimental results section. The issue with these methods is that a reference mesh needs to be defined that is with respect

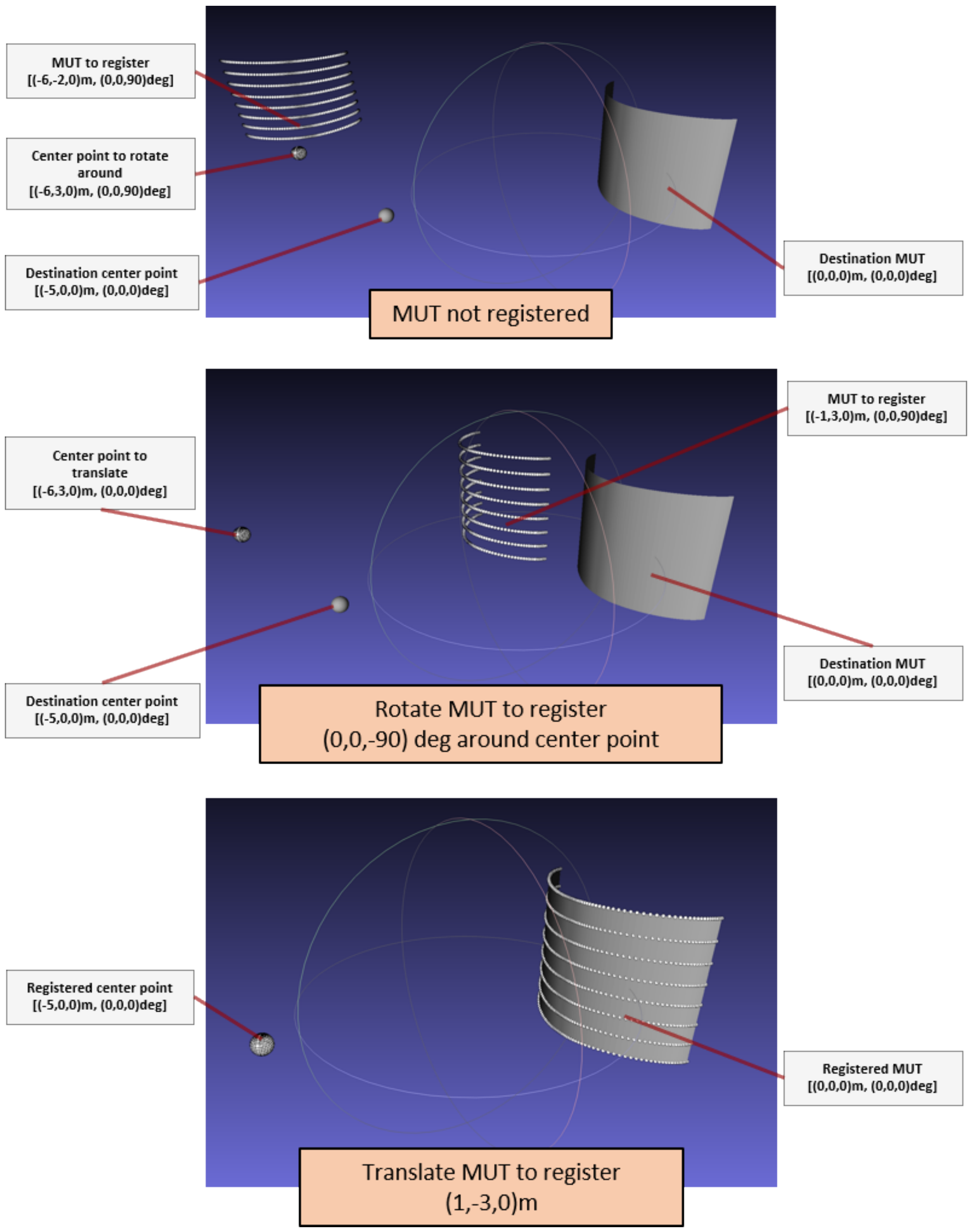

MUT to register [(-6,-2,0)m, (0,0,90)deg]
Center point to rotate around [(-6,3,0)m, (0,0,90)deg]
Destination center point [(-5,0,0)m, (0,0,0)deg]
Destination MUT [(0,0,0)m, (0,0,0)deg]
MUT not registered
Center point to translate [(-6,3,0)m, (0,0,0)deg]
MUT to register [(-1,3,0)m, (0,0,90)deg]
Destination center point [(-5,0,0)m, (0,0,0)deg]
Destination MUT [(0,0,0)m, (0,0,0)deg]
Rotate MUT to register (0,0,-90) deg around center point
Registered center point [(-5,0,0)m, (0,0,0)deg]
Registered MUT [(0,0,0)m, (0,0,0)deg]
Translate MUT to register (1,-3,0)m

Figure 3.6: Center point transformation example

to the workspace. If that is not clearly defined, then there will be an improper virtual-physical correspondence with the mesh. It could also lead to improper alignments of meshes, depending on the definition of geometries seen later in figure 5.16. Also, these methods are less accurate results then center point transform due to Root Mean Square (RMS) approximations comparing boundary overlapping of the point clouds, which determined tranformation vectors required for registration. Each approximation attempts to decrease the RMS value by selecting "random" transformation values towards the frame to align. Sometimes if there is not a proper geometric setup for proper registration, for example if two flat images are obtained with no geometric similarities, then registration will be very flawed. For demonstration purposes, figures 3.7 and 3.8, where two stereo camera shots were taken without any transformation information of where the camera was with respect to the sample and the processing was done though CloudCompare. In figure 3.7, the manual selection of reference points for registration were the holes going through both car piece samples, where the two images obtained were from the front and back of the sample.

Using stereo-vision techniques, there are some effects that may misshape a single image. If the camera is not within an effective range of detecting the object, such as if the camera is either too close or far to the object, then the point cloud of interest will lose points and become deformed. For example, the Intel RealSense D425i stereo camera used for this report, the ideal range for acquisition is between $0.3m$ to $3m$ meters. Generally, the further the object is with respect to the camera will yield lower resolutions of the MUT reconstruction. It was observed with the setup that any component in the environment is touching the MUT, such as if the MUT is laying flat on the ground, there is a morphing phenomena between the two objects. Due to this, it is highly preferred to have the MUT on a placement where these effects are not obtained by the stereo camera. It was also observed that certain lighting effects on the MUT would cause improper or missing information, creating holes or warps on the surface. Transparency of the items within the physical environment also tend to have effects, where fully transparent objects will not appear in the obtained image. Most warping effects or missing points, assuming not very significant, can be filtered during the reconstruction process. It would be more preferred to obtain clean images on acquisition to eliminate noise. Analyzing and deploying these methods are possible in a future report.

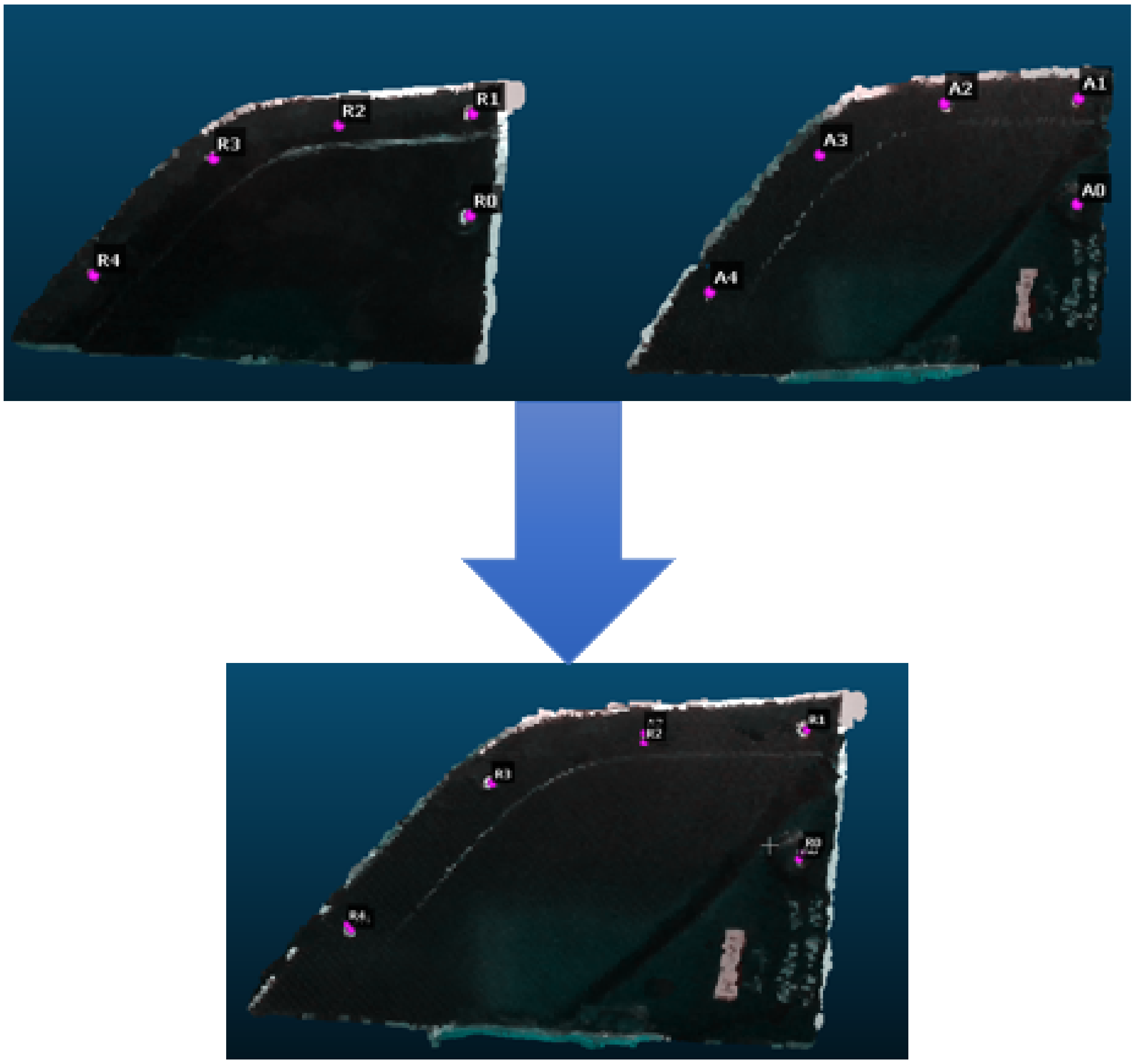


Figure 3.7: Manual point pair selection using a car piece

3.3 MUT Surface Reconstruction

Once the image is acquired and aligned with respect to the workspace, then reconstruction is preferred as the final process before toolpath generation. It is possible for the reconstruction step to be done before registration assuming a single image is used, depending if the acquisition allows for mesh outputs. For multiple images however, a cluster of vertices of each point cloud will be the output of registration rather than a mush of meshes with interlacing faces, which

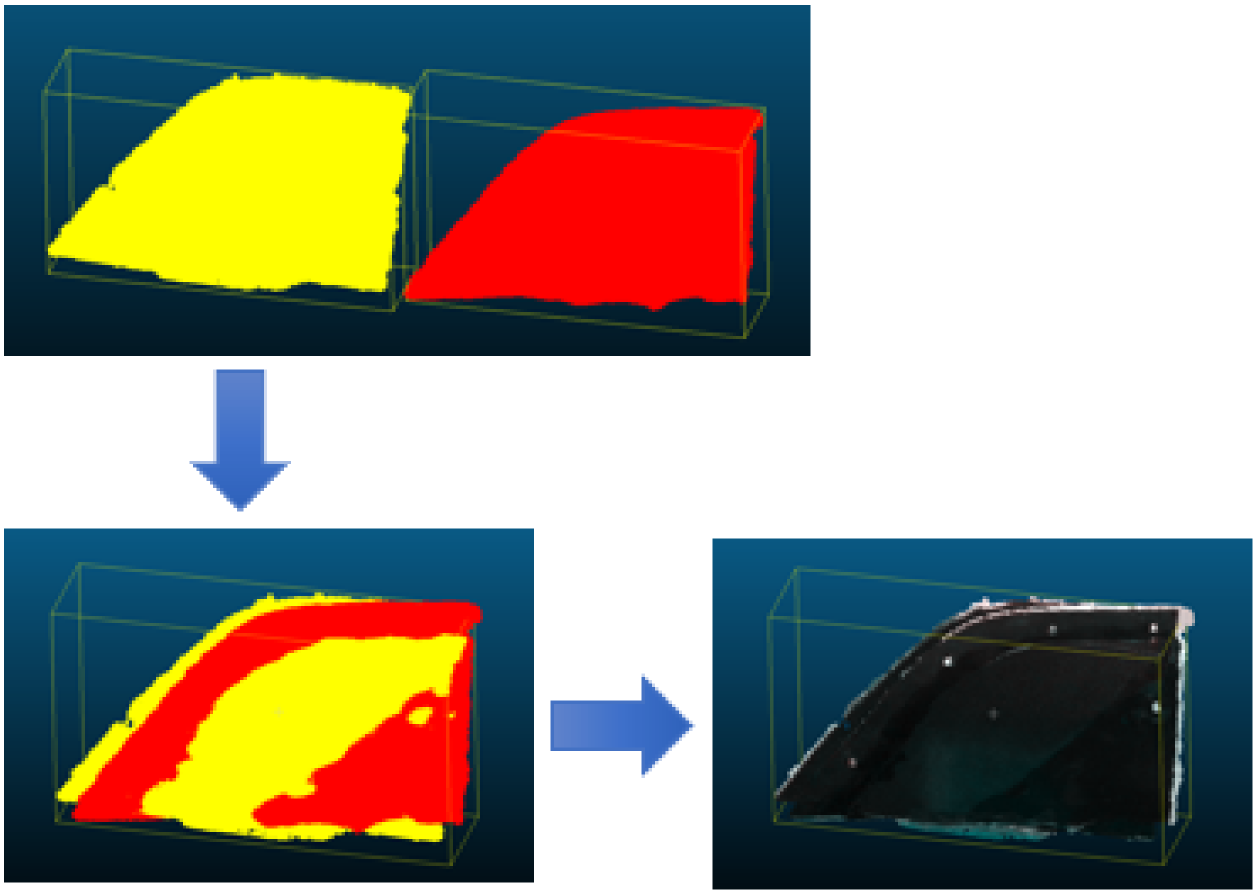


Figure 3.8: Automatic fine registration through iterative closest point with a car piece

is unfeasible for path planning. Hence why reconstruction is regarded as the last step. The reconstruction process of the MUT is analogous towards the reconstruction and visualization of NDE data. Reconstruction from registered point clouds have been registered and combined to one image requires the following processes: clean the merged point cloud, reconstruct the mesh, remove any boundaries or irregularities from reconstruction, smooth or otherwise improve the quality of the mesh. To clean the image point clouds, background information or other oddities within the image must be removed so the only features are within the MUT point cloud cluster. These point cloud cleaning measures are done through CloudCompare. A Statistical Outlier Removal (SOR) filter is useful for immediately removing single or small clusters of a points that are not attached to the MUT cloud. In figure 3.10, a CAD-defined curved surface was generated which included vertices of various distances around the point cloud of interest. Simply applying the filter removes

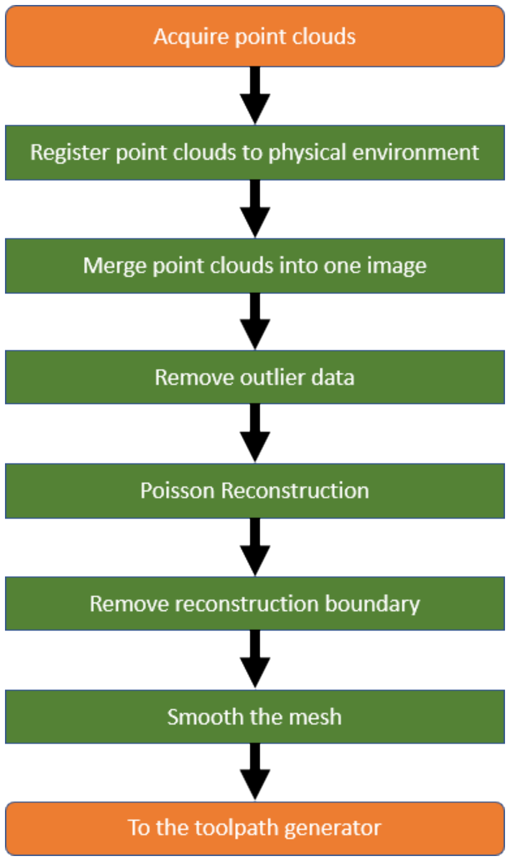

Acquire point clouds
Register point clouds to physical environment
Merge point clouds into one image
Remove outlier data
Poisson Reconstruction
Remove reconstruction boundary
Smooth the mesh
To the toolpath generator

Figure 3.9: Reconstruction method using multiple point cloud images

most of these considerably erroneous points. While this filter is effective for filtering small and independent regions away from the MUT, it is not effective towards removing larger background or touching components. Eliminating these large clusters were done manually through point removal or segmentation tools in CloudCompare for the sake of this report, and automating this process is possible for a future topic. It is also likely that discrepancies due to acquisition errors need to be manually removed on the MUT, which becomes more likely there larger set of registered images are used. These tend to add bumps or odd shapes on top of the surface which are not properly removed through SOR filtering or smoothing and are currently extracted by manual point removal.

Once the point cloud has been cleaned, Poisson surface reconstruction is used through Cloud-Compare using Dirichlet boundaries [11] [10]. The output will be a mesh of the MUT with somewhat spherical boundaries and a Scalar Field (SF) which provides a histogram of output density information applied to the reconstruction algorithm. There is a parameter to select the

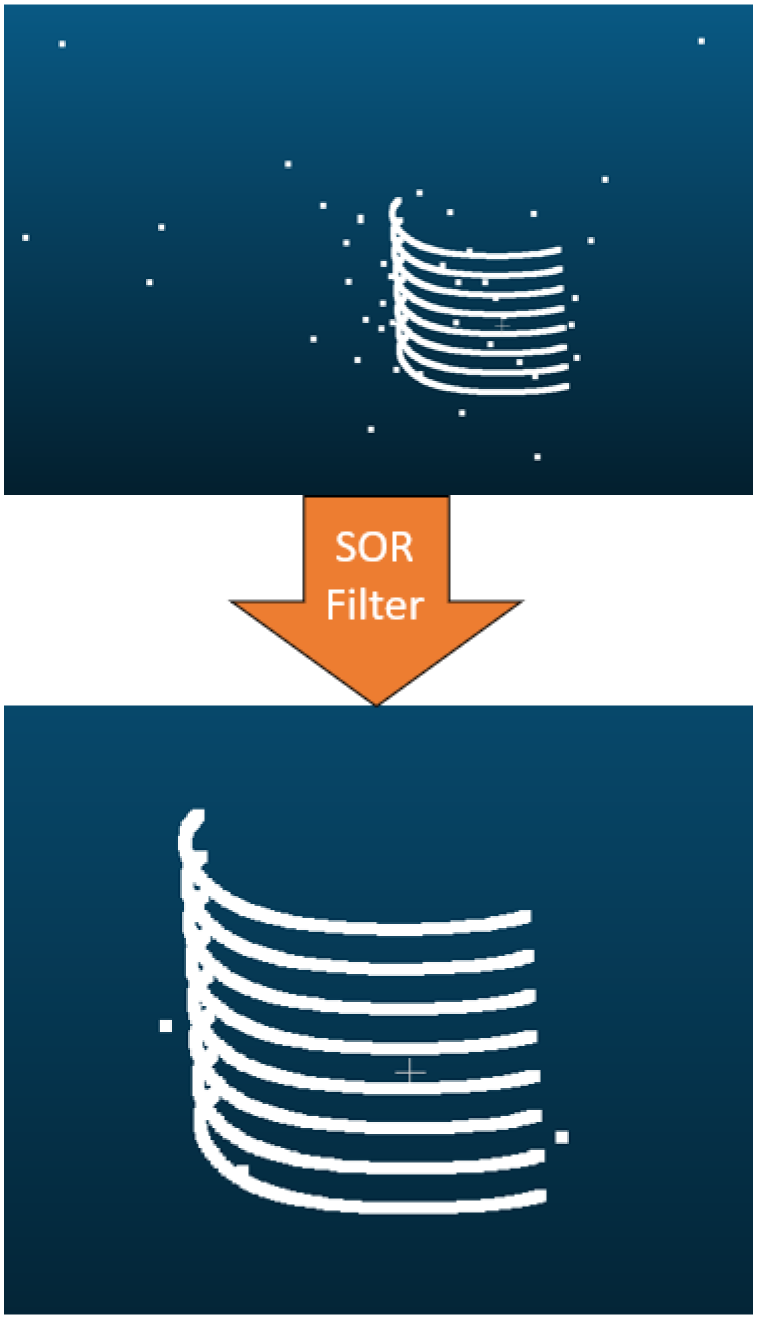


Figure 3.10: SOR filter example in CloudCompare

octree depth of the reconstructed model, in which the lower value will provide faster, smoother, and less resolute results while larger values provide slower, rougher, and more resolute results. The reconstruction will also have a boundary that wraps behind the mesh that must be removed for path planning. To remove this boundary, the range within the SF output density histogram should be selected to closely match the maximum curve of density information within the histogram. Simply put, the closer the range values are towards this curve, the more boundary edge information is eliminated. The closer this range value is to the curve will also increase the possibility of removing key features of the reconstruction. It is therefore preferred to balance this range parameter so it removes the boundaries effectively without interfering with the reconstructed model. Due to these complications, currently these range values are manually approximated within the CloudCompare GUI, but this process can be automated in the future. In figure 3.11, the curved surface used before was reconstructed using Poisson reconstruction at an octree depth level of 8 using Dirichlet boundaries. The surface was also subdivided at the midpoint with three iterations to remove the large gaps seen in the the original model. The scalar selection feature was used to remove most of the boundaries around reconstructed model. The surface of the model was still considerably "rough" and had an overhanging boundary that slightly extended past the MUT. It is also possible to use manual point removal methods while using a simpler boundary method such as Neumann to remove the boundaries.

Once the range is selected and the boundary is filtered out, the model is exported to MeshLab for final mesh cleaning. As the mesh after Poisson reconstruction tends to generate bumps along smoother surfaces, mesh smoothing methods are preferred. The method selected was Laplacian smoothing is used, as it allows for encoding of local vertices to improve the quality of the model [13]. It also allows for control of smoothness by selection of number off iterations of the algorithm. Once smoothed and cleaned, the reconstructed model of the model should be ready for toolpath generation. Figure 3.12 shows the smoothing process using ten Laplacian smooth iterations on a reconstructed model that didn't have subdivisions, which created bumps along where the vertices were placed. The approximation allowed the reconstructed model to more closely approximate the

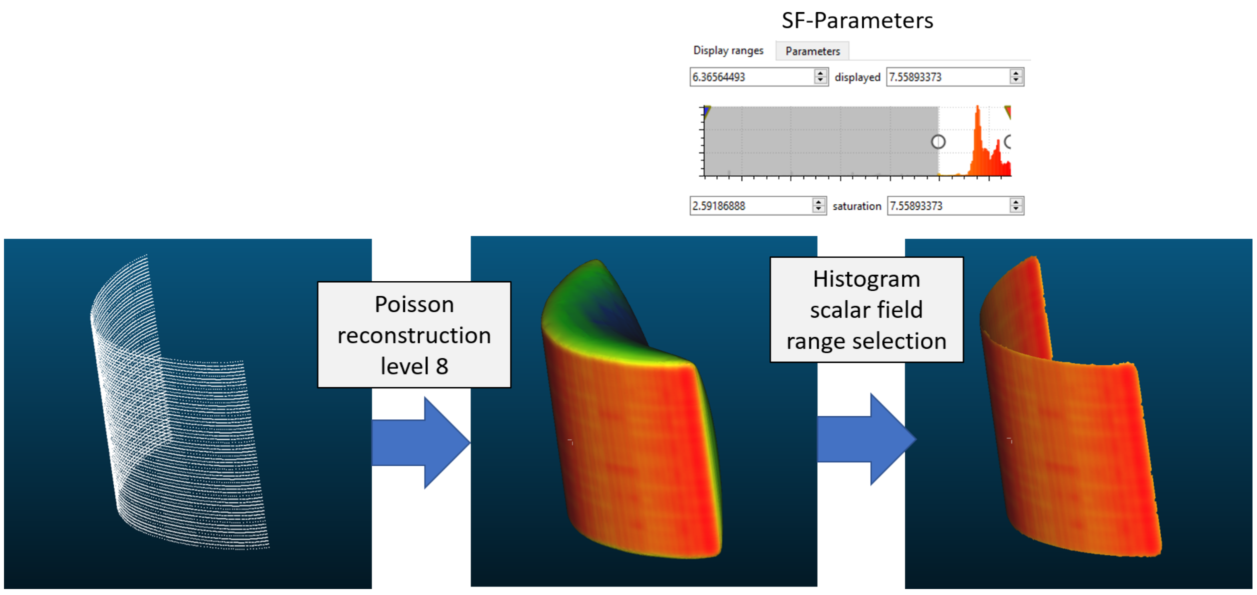


Figure 3.11: Poisson reconstruction example in CloudCompare

shape of the goal model.

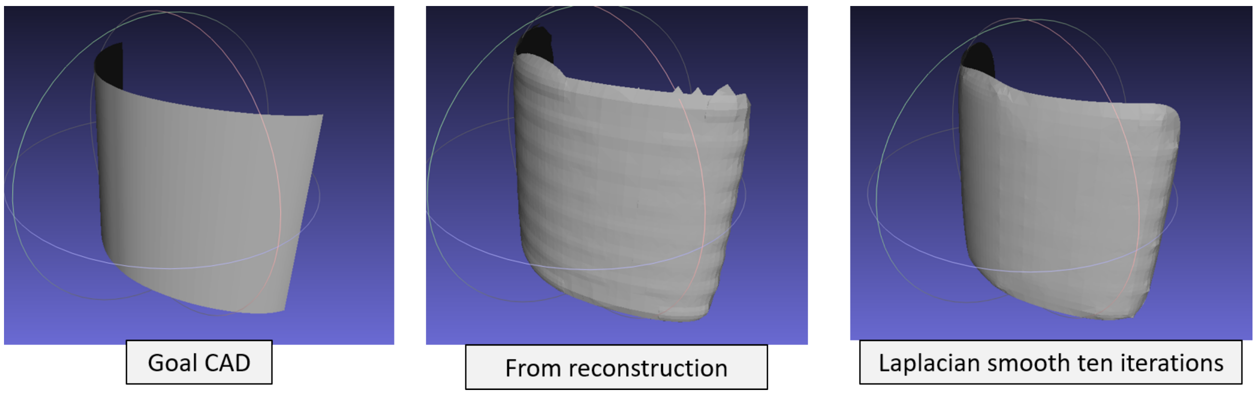


Figure 3.12: Laplacian smooth example in MeshLab

CHAPTER 4

NDE SCANNING CYBER-PHYSICAL SYSTEM

The NDE scanning cyber-physical system refers to the functionalities of performing a scan alongside a desired surface. It is assumed at this point that the environment including the MUT's surface has been reconstructed or is at least known. The main operations this system must conduct is to generate a toolpath, simulate actuation to confirm DH joint movements around the environment, send the list of movement commands to the physical robots for actuation, and obtain then synchronize pose data of the sensor with the set of NDE data obtained at the same time. Note this report only focuses on reconstruction and path planning, so simulated and physical movements are only theoretical for future reference and have not been tested. In any case, these functions are to be explained in detail the next sections.

4.1 Toolpath Generation: 5D Surfacing

The goal of this system is to obtain a toolpath in which the end-effector of an actuation system moves along the surface of the MUT. Two explored methods were to use the licensed software SprutCAM to generate toolpaths from a known MUT and it's environment, and a customized OpenGL-based workbench that generates toolpaths from ray-triangle intersections. The output of the toolpath should be a list of waypoints where the end-effector reaches from the start to the end of the scan. Each waypoint contains a translation T_{xyz} and a rotation R_{xyz}. As rotations of the end-effector can be simplified two dimensions $R_{\alpha\beta}$, only five dimensional parameters are required for translation and rotation. This is effective for simplifying the movement requirements for the robots, however for calculations R_{xyz} will still be referred to. The definition of where these points remain depends on the configuration of the scan. For example in a $2D$ raster scan, only knowledge of the width and height of a $2D$ sample, the boundaries of the $2D$ raster scanner or gantry, and the desired liftoff are required. In this case, a $2D$ plane of potential points can be placed in regards to the resolution or number of points required due to the boundary restrictions of the configuration. Since $5D$ surfacing requires more complex information, configurations may vary. Different configuration may effect the placement of waypoints or the path to reach each waypoint.

Two classifications of points are considered. One is the set of unorganized waypoints, or targets where the probe should reach during a scan. These waypoints are to later be organized using traveling salesmen algorithms. It is also to initially organize points in a default configuration such as zig-zag. These lists of points will simply be called the waypoints set. The other is the organized set of waypoints, which is considered as the toolpath or scanpath. Waypoints are generally on the surface of the specimen where the probe will move towards and obtain data, but some waypoints may not be on the surface. For example, the placement of the robot's end-effector starts and ends is considered as a safe waypoint or target. These start and stop waypoints are always determined to be at the start and end of the toolpath list respectively. There are also jump waypoints, where if the path between two points is obstructed, the probe may "jump" away from the surface in order to avoid the obstruction. Jump waypoints are generally determined during the path generation process rather than defined within the unorganized waypoints set. These jumps typically occur during SprutCAM's toolpath generation processes, and have yet to be implemented within the customized toolpath generator due to it's complexity. In any case, it's important to classify waypoints as "on the surface" and "not on the surface" since the probe's recorded data should only be along the surface within a desired lift-off rather than solely in the air. Points that are acquired while in the air should not be recorded, or otherwise need filtering methods to be removed.

Toolpath generation has two operations. The first process is to generate the waypoints set required for the scan using the surfaces profile and other requisite safe or jump waypoint. The second operation is to organize the waypoints set to where the probe will reach from the start to end of the scan. In other words, create the toolpath S. It is required that the virtual space for computation of the toolpath contains the CAD model of the MUT to scan and the actuation system with all DH parameters provided. Actuation systems parameters and kinematic calculations are handled later while simulating scans using RoboDK. The virtual space must be aligned with the physical workspace as well, which should be handled through $3D$ reconstruction or other alignment methods with a predefined model. Validation of this virtual-physical interfacing this is not perfect in this report and will be handled later. The toolpath will output an array of transformation parameters as waypoints, or S, which contains all the forward kinematic movements required for the end-effector to move along the MUT's surface. The probe which will be placed on the end-effector should be normal along side the MUT's surface at all times during the scan, save for any movement's the probe is in the air. It is also preferred to have a constant standoff between the MUT's surface and end-effector in order to make up for any errors of alignment of the CPS to prevent collisions. This standoff should be constant in order to prevent positional discrepancies between NDE data at different points.

4.1.1 Basic Generation Algorithms

Before further explanation, basic generation algorithms are described in order to provide a better idea for future concepts. To start off, a $2D$ configuration is considered. Figure 4.1 shows a set of waypoints alongside a $2D$ flat MUT's surface using a grid-like configuration. Each waypoint is along the surface of the MUT in $2D$ space, therefore having $T_{xyz} = [x, y, 0]$ and $R_{xyz} = [0, 0, 0]$. The boundaries where waypoints may be defined are within l_x and l_y. Both must be less than order equal to the width w and height h of the sample itself respectively. This definition assumes only positive values for lengths and start points, or the scan environment is between $E_{min} = [0, 0, 0]$ and $E_{max} = [w, h, 0]$. l_x and l_y also depend on the start point of the scan, $start_x$ and $start_y$. Therefore, $w \geq l_x + start_x$ and $h \geq l_y + start_y$ in this configuration. Lengths may also be negative in which it's direction is flipped, which direction is defined by the coordinate system used. Another coordinate system as the middle of the scanning environment as $[0, 0, 0]$ with limits $E_{min} = [-w/2, -h/2, 0]$ and $E_{max} = [w/2, h/2, 0]$ To define the number of points alongside l_x and l_y, resolutions r_x and r_y can be set as an input to the system. In this case, $r_x = 4$ and $r_y = 3$. Having the resolution known, the step-sizes s_x and s_y between points are found by using $s_x = \frac{l_x}{r_x}$ and $s_y = \frac{l_y}{r_y}$. Inversely, the step-sizes can be used as an input to solve for the resolutions $r_x = l_x s_x$ and $r_y = l_y s_y$. Figure 4.1 contains labels to describe the height and width of the sample (red), the boundary lengths (blue) and point step-sizes (magenta) the along the x and y axes, the waypoints (yellow) including the start point (cyan), and all possible paths (green).

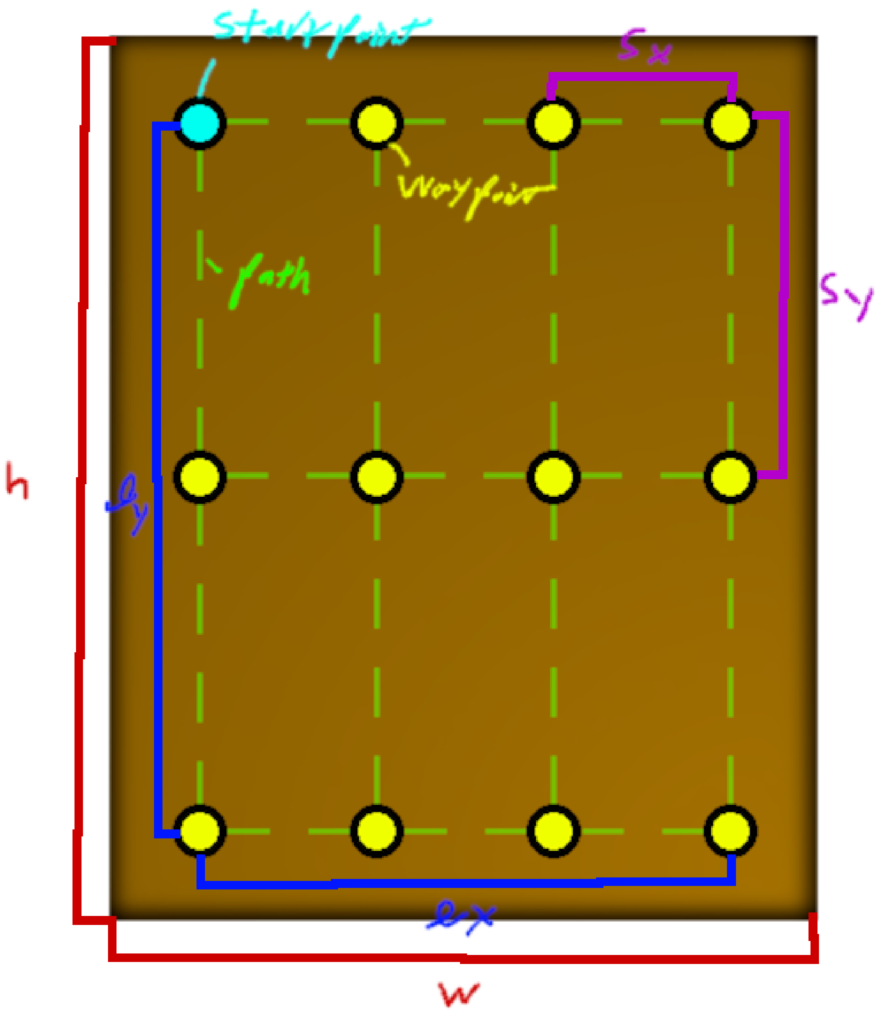


Figure 4.1: Unorganized waypoint example on a 2D MUT

Points may be regarded as "reached" or "not reached" throughout the scan. It is possible for points to be in a "not reached" state at the end of the scan, depending on the complexity of the toolpath generator. For example, it would be advantageous to include collision detection methods to avoid obstacles within the path to prevent touching the sensor. These methods add complexity which increases both computation time and programming time. Because of the extra complexity which is mostly mitigated through the use of larger liftoff distances, obstacles are ignored in this report except for the usage of RoboDK.

Algorithm 4.1: Line-by-line point generation

```
Input: Two real numbers $l_x$ and $l_y$, two non-negative integers $r_x$ and $r_y$
Output: List of points, $list$
    $s_x = l_x / r_x$;                                          ▷ Define sizes
    $s_y = l_y / r_y$;
    $list = new\ PointsList(s_x * s_y)$;          ▷ Define as $PointsList$ with capacity $s_x * s_y$
    for $x = 0; x < r_x; x = x + 1$ do                     ▷ Iterate through all lines along $l_x$
        $pos_x = s_x * x$;                                  ▷ Define the position on x
        for $y = 0; y < r_y; y = y + 1$ do          ▷ Iterate through point on the line along $l_y$
            $pos_y = s_y * y$;                              ▷ Define the position on x
            $point = new\ Point(tx = pos_x, ty = pos_y, tz = 0, rx = 0, ry = 0, rz = 0)$;
            $list.Add(point)$;                     ▷ Create and add a point to the list
        end for
    end for
    $return\ list$;
```

The most simply way of organizing ways points is to conduct a line-by-line scan, which is typically a default configuration. This scan can be generally done to generate all the points required for a scan then use a path planning algorithm to organize the points. First, define a *Point* class with inputs tx, ty, tz, rx, ry, and rz with constructor $Point(tx, ty, tz, rx, ry, rz)$. Also, define a list of points at $PointsList$, where the constructor looks like $PointsList(size)$ and a function to add a point is $PointsList.Add(new\ Point(tx, ty, tz, rx, ry, rz))$. The algorithm for a line-by-line scan looks somewhat like this is provided in Algorithm 4.1, the resultant shown in figure 4.2. In this configuration, the total path distance δ_{ll} is defined as the summation of all l_y components plus the summation of each hypotenuses, or $\delta_{ll} = (r_x * l_y) + [(r_x - 1) * \sqrt{s_x^2 + l_y^2}]$. Note that distance is a simplified way to compute the total "weight" of the path. It may also be noted δ is not dependent on r_y and s_y for this scenario, meaning that placing extra waypoints alongside l_y does not effect the distance. It is important to consider these points along l_y when considering $3D$ solutions, as depth changes will occur and sampling these positions are important if trying to find a best distance or weight solution.

While the line-by-line scan is easy to implement, there are better options available to generate the points with a decent distance and low computation. The method used for generating points is to place points within a zig-zag pattern instead of a line-by-line pattern. The algorithm is similar to the

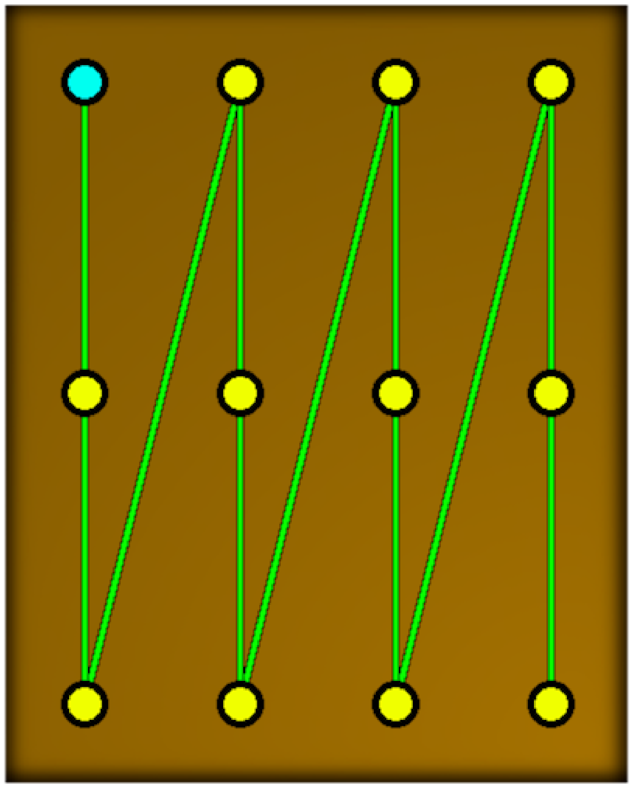


Figure 4.2: Line-by-line waypoint example on a $2D$ MUT

line-by-line pattern one major change, which is that the points along l_y will flip for each even/odd instance. To apply this, a boolean value zig is required to mark these changes. This change is seen in Algorithm 4.2, and visualized in figure 4.3. If zig is true, then the path along l_y will move in a positive manner, while if zig is false, then it will move in a negative manner. The position along l_x will always move in a positive manner. The total distance δ_{zz} will now be the summation of lines along l_y and summation of s_x parts, which in this scenario is just $(r_x - 1) * s_x = l_x$. Therefore, $\delta_{zz} = (r_x * l_y) + l_x$ for the $2D$ zig-zag configuration. Compared to the line-by-line scan, the difference between the two is $\sqrt{s_x^2 + l_y^2}$ and s_x. As $s_x \leq \sqrt{s_x^2 + l_y^2}$, $\delta_{zz} \leq \delta_{ll}$ and $\delta_{zz} = \delta ll$ only when $l_y = 0$, meaning only a single straight line along l_x for both configurations. Because of this inequality and the near non-existent difference in computation, the zig-zag configuration is desired over the line-by-line configuration. Just like with the line-by-line example, r_y and s_y are not needed, but are important for $3D$ configurations as shown later.

When considering a $3D$ model, more points alongside l_y is favorable since it create a finer

Algorithm 4.2: Zig-zag point generation

Input: Two real numbers l_x and l_y, two non-negative integers r_x and r_y
Output: List of points, $PointsList$

```
$zig = true;$                                         ▷ Define the zig-zag variable
                         ▷ Note: zig => zag = true, and zag => zig = false

$s_x = l_x / r_x;$                                    ▷ Define sizes
$s_y = l_y / r_y;$
$list = newPointsList(s_x * s_y);$                    ▷ Define as $PointsList$ with capacity $s_x * s_y$
for $x = 0; x < r_x; x = x + 1$ do                    ▷ Iterate through all lines along $l_x$
    $pos_x = s_x * x;$                                ▷ Define the position on x
    for $y = zig \, ? \, (0 : r_y - 1); zig \, ? \, (y < r_y : y >= 0); y = y + zig \, ? \, (1 : -1)$ do
                                                      ▷ Iterate through point on the line along $l_y$
        $pos_y = s_y * y;$                            ▷ Define the position on x
        $point = new\ Point(tx = pos_x, ty = pos_y, tz = 0, rx = 0, ry = 0, rz = 0);$
        $list.Add(point);$                            ▷ Create and add a point to the list
    end for
    zig != zig;                                       ▷ Flip the zig->zag or zag->zig
end for
$return\ list;$                                       ▷ Return the resultant list
```

path alongside curved surfaces and has a better than of detecting the location of a sharp geometric change alongside the z-axis. Figure 4.4 shows the way points, where the darker regions are placed "into the screen" and the lighter regions are placed "out of the screen". A $3D$ mesh representation of this is also shown in figure 4.5. These figures will come later into play when describing the traveling salesmen problem later.

Other configurations for point generation are possible. For example, an hemispheroid path generator was created in order to study circular movements away from a sample, or to be used with simple sphere-like objects. The path generator would create an array of $2D$ ellipses that would "decay" alongside a defined sizesize along z. The results of this were shown previously in figure 3.4. It would also be advantageous to create a reference frame where the planar path-planner generated from either the line-by-line or zig-zag configuration may be transformed, which would be useful for obtaining a precise $3D$ model of the environment using a sensor such as a high-precision distance detection sensor that outputs only one scalar data rather than an image. Such would be a preliminary scan from a approach direction desired that would run a planar scan away from the

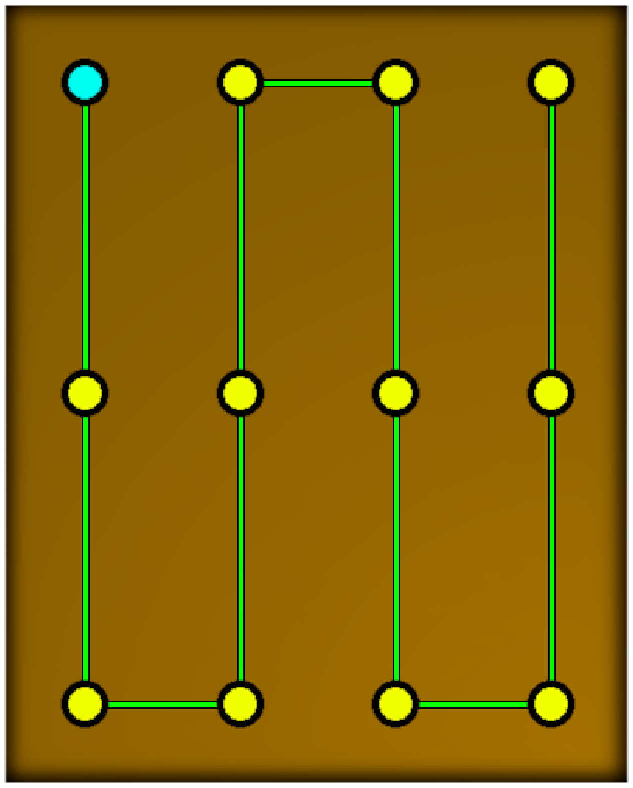


Figure 4.3: Zig-zag waypoint example on a $2D$ MUT

sample to obtain the $3D$ model used for path planning. It could also be used to simply scan flat panels without computational or development overhead using a complex actuation system.

4.1.2 Custom path planner

4.1.2.1 Ray-Triangle Intersection Array Toolpath Generation

In order to control the properties of each waypoints and it's containing set, a OpenTK-based CAD workbench program was developed which manages the visualization and generation of waypoint sets and it's organization into a toolpath. OpenTK is an open-source toolkit which allows for low-level controls for OpenGL bindings. SimpleScene, open-source OpenTK implementation with various basic functionalities of OpenTK into a workbench as a $3D$ scene, was used as the base of this program [9]. The importance of developing such interface rather than a typical command line is the confirm each waypoint coordinate alongside an input mesh before placing inputs into an inverse-

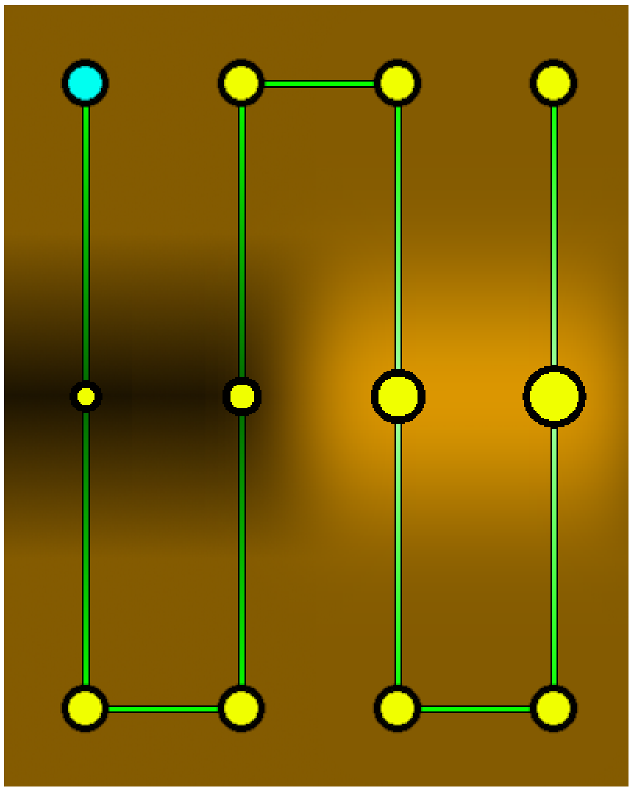


Figure 4.4: *Z*ig-zag waypoint example on a 3*D* MUT

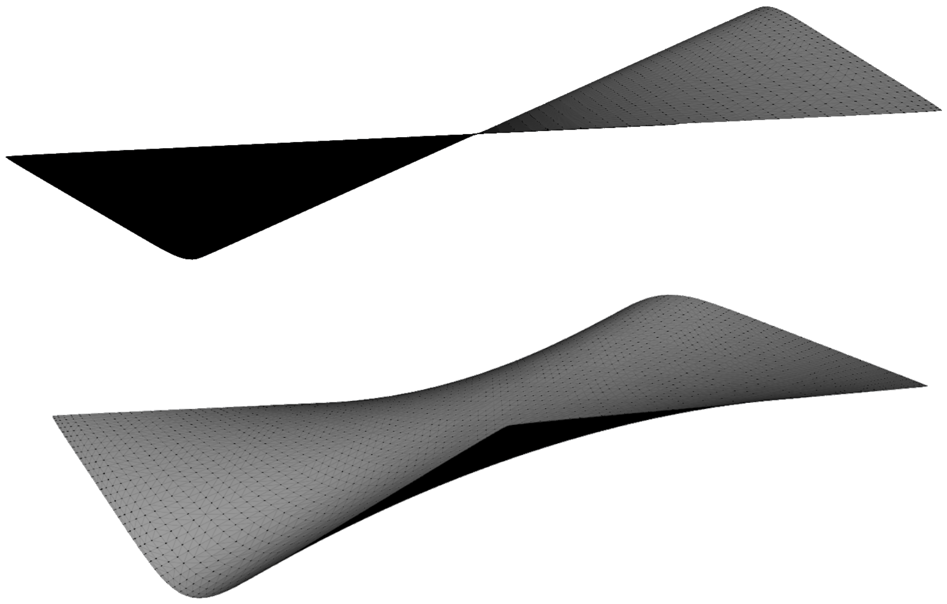


Figure 4.5: Mesh representation of figure 4.4

kinematics solving software, such as RoboDK, while also main low-level control of actions on the mesh. The low-level control allows for programmatic interaction towards each element of a mesh, including each face on the mesh. By design, several different toolpath generation method utilities and techniques can be developed in the future. The downside is that knowledge and management of lower level functionality can be burdensome for development of newer features. For the focus of this report, a ray-triangle intersection algorithm was developed within this program. In the future, other algorithms might be possible such as plane versus mesh edge intersection.

The theory of the ray-triangle intersection algorithm requires a build up of information about the MUT as a mesh and ray-triangle interaction. A mesh within perspective of a low-level program is made up of vertices holding only translational T_{xyz} information. Mesh edges and faces are found based on the formatting of the input mesh file loaded into the program. Every face on a mesh is made up of a flat triangle shared between three vertices, where each face is single-sided which direction is determined through face culling [5]. If a face has more than 3 vertices, then it is still described as a combination of triangular faces. An example of this is shown in figure 4.6, where a sphere with it's wireframe is shown to only be made up of triangular faces. The correct direction of each face is important as it determines the proper orientation of the normal vector of the face, as well as proper rendering of the mesh. If the face is culled in the wrong orientation, then the face normal will be inverted which is unwanted for path generation. Mesh edges might be an important consideration for other toolpath generation processes, but will not be considered for this report. Such would be useful for "continuous" scanning alongside l_y, which is explained later with a plane-mesh algorithm.

The next object within a $3D$ environment is a ray, which is described as a line in space with either a finite length by being bounded between exactly two points or with infinite length if one or both points are not available. Rays are analogous of edges on a mesh in the sense that it requires two vertices to render, however rays are not a part of a $3D$ mesh. Rays can still be represented in $3D$ space, however. In the algorithm, the ray is determined to be one sided, so the order of points is important or in other words, a starting point will "shoot" out the ray until it reaches the end point. The interaction of collisions between a ray and a mesh is determined by the splitting up the mesh into a set of triangular faces, and comparing the if there is a collision between the ray and any

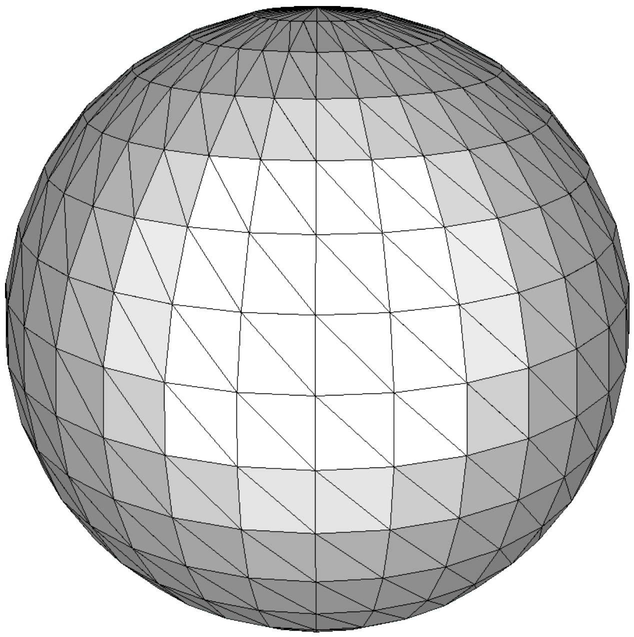


Figure 4.6: Sphere mesh made up of triangular faces

part on the mesh. The Möller–Trumbore intersection algorithm is used to perform each individual ray-triangle intersection task [12]. If there is a collision, then the top-most intersection point, or the point with the shortest distance to the start of the ray, is recorded. The intersection algorithm will return it's translational parameters in T_{xyz} and the rotation of the ray with respect to global space is also know. The normal vector of the triangle as a face of the mesh is required, however. This normal is determined by the direction of the triangle from face culling. Therefore two modes are possible: return the orientation of the ray itself as R_{xyz} of the collision or intersection point, or return the normal of the face. From these two parameters, the position and rotation of a point is known. If the ray does not intersect with and faces on the mesh, then a point is not generated. For optimization in the case that a mesh has a large number of faces, ray-hull intersection is done and passing this test is a prerequisite before the algorithm checks the ray against every face on the mesh. An example of ray-triangle interaction is shown in figure 4.7, where the red intersection component shows the rotation after face culling, where the z-axis is along the normal.

With several waypoints that are generated from the ray-triangle approach, it is possible to create a waypoint set and toolpath by generating multiple rays against the mesh. While the generation

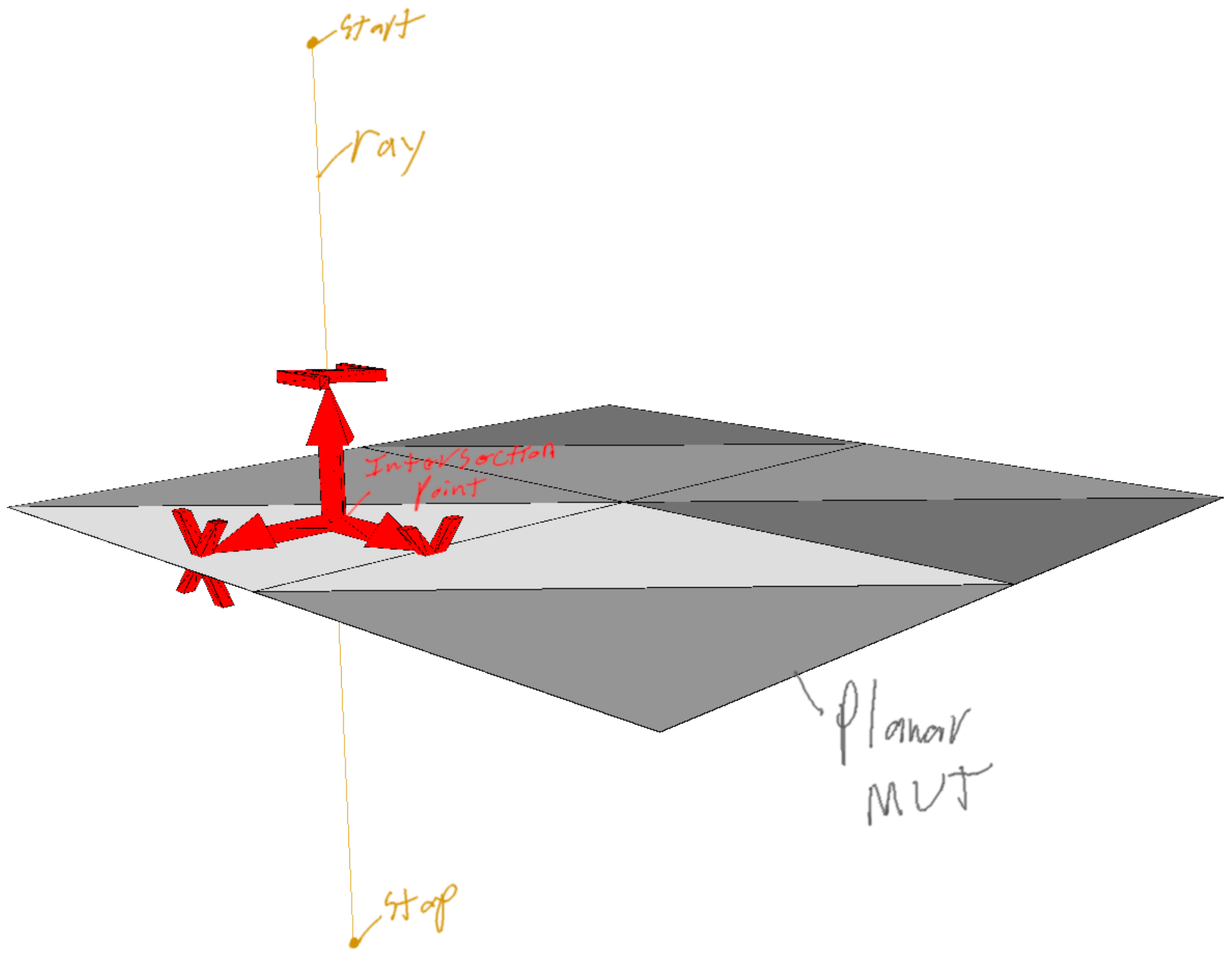


Figure 4.7: Ray-triangle intersection against a flat planar sample

of these rays may occur is several different patterns, the simplest is to create a $2D$ array of rays alongside a defined bounded $2D$ plane. An example of this against a unit plane is shown in figures 4.8 and 4.9. The bounds on this plane can be simply regarded in terms of a definable width and height. As the rays on the plane also have a length, then the boundaries where the rays hit generates a "rectangular prism" away from the starting plane. The number of points along the plane are determined as resolutions r_x and r_y with orientation respect to the rectangular prism. With larger resolutions, more points will be generated which increases the time to both conduct the scan, as well as time for toolpath organization computation and post processing. Larger resolutions also provide more information about the specimen which may be required, so a balance to determine the best resolution should be determined. When the rectangular prism of rays intersections with a mesh, assuming conditions such as proper face orientations are set, then a set of waypoints are be placed onto the surface of the mesh. Any rays that do not intersect will not generate waypoints. The rectangular prism can be transformed alongside space, where the position of the starting plane is considered as the "approach". For objects that require scans around the sample rather than a facial scan from one angle, it is possible that one approach is not sufficient, hence why it might be advantageous to append several toolpaths generated from different approaches. Other approach methods can be possible to hopefully optimize such cases, possibly a spherical approach method could be determined for the future.

For further clarification on figures 4.8 and 4.9, a grid of rays (thin and orange) is projected from the approach plane (in green) with a resolution of $r_x = r_y = 5$ points and lengths $l_x = l_y = 0.5$ units starting at the center of the a unit planar MUT (dark grey) with $w = h = 1$. And end plane (light grey) is also displayed, which shows the end points of the rays. Figure 4.8 shows what the rectangular prism (purple) would look like in this scenario. Intersection points lay normal to the surface of the MUT, which starts from red to blue. The path along these points was set to a zig-zag pattern (wide and orange), which is more clearly seen in figure 4.9. Each waypoint is normal to the surface, which might look slightly skewed due to the projection of the CAD viewer being in a perspective mode rather than an orthographic mode.

There is are numerous configurations that can be used with the approach, depending on the geometry of the input mesh and sizes available. For example in figures 4.10 and 4.11, the approach was tilted 45° along x, and shifted to meet at the center. Other configurations are the same as the previous unit plane examples. The notable differences are that the points shift and separate due to different approach angles, and that the intersection points are still normal to the surface despite the change in each approach angle. Another example given is uses a "soccer ball" model, where the ball has a radius of 2 unit and each rotation conducted is 45° around the model, shown in figures 4.12 and 4.13. The configuration was set to generate $r_x = r_y = 9$ points with lengths of $l_x = l_y = 2$ units path alongside the top of the ball which would generate a zig-zag pattern. This demonstrates that the normal of each intersection point will adjust toward the rotations of the model. There are a

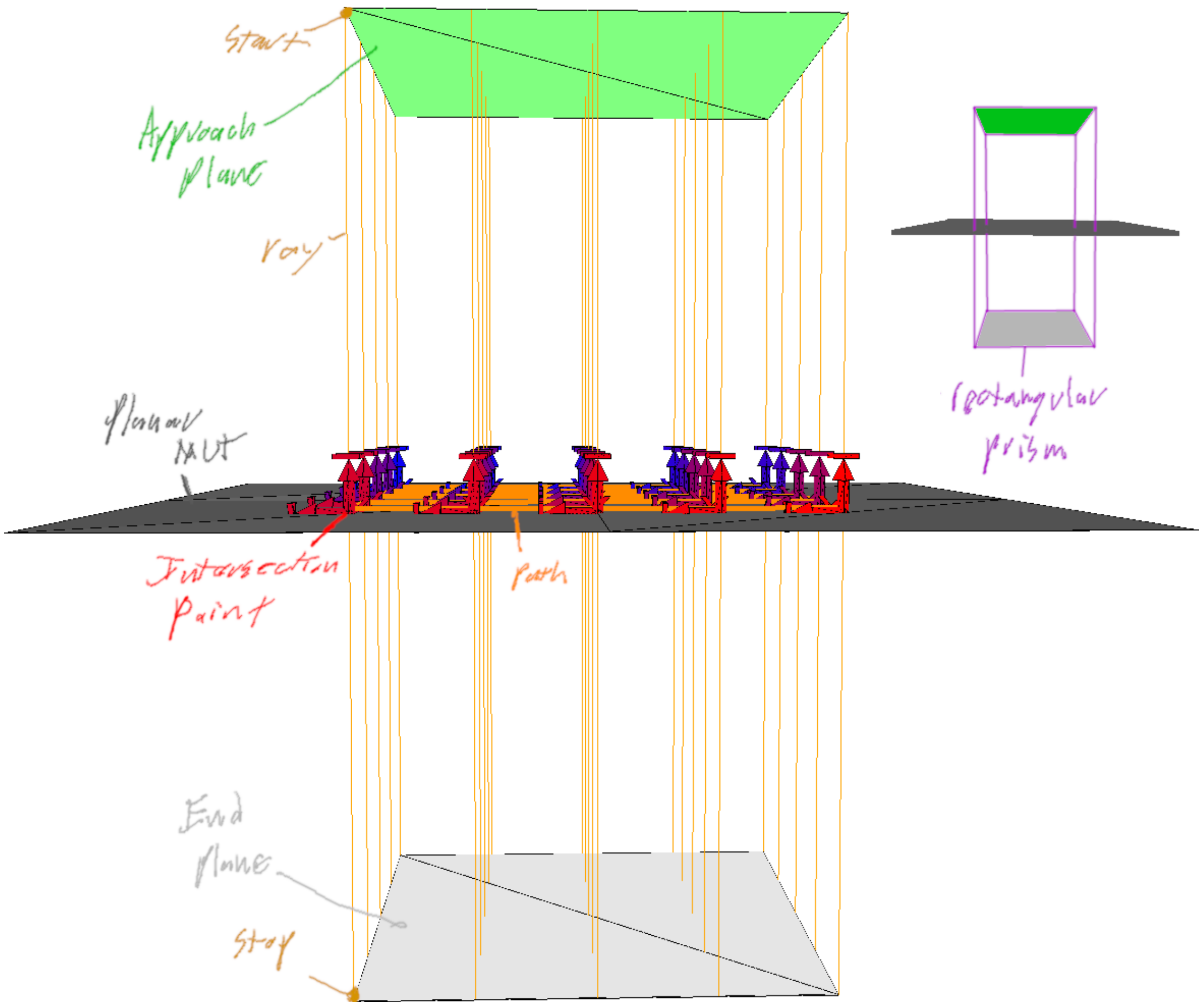


Figure 4.8: Plane-triangle example against a unit plane MUT showing the approach planes

couple considerations from this example. First, it might be advantages to keep a consistent rotation along z on the end-effector, however rotation along z changes while moving along the ball, hence why static rotations are implemented. Static rotation parameters were not used in the experimental results as they were not needed. Also when a point is lays on top of an edge, it should be "normal to the edge", not normal to an adjacent face as it is seen in the figures, however this requires a ray-edge intersection implementation that obtains a proper normal. Finally, as shown in figure 4.14 where the resolution is set lower to $r_x = r_y = 8$ that the paths collide inside the mesh itself, so path planning around these collisions should be developed. Ray-edge detection and improvements on path-planning in regards to self collision should be implemented in the future, however they should

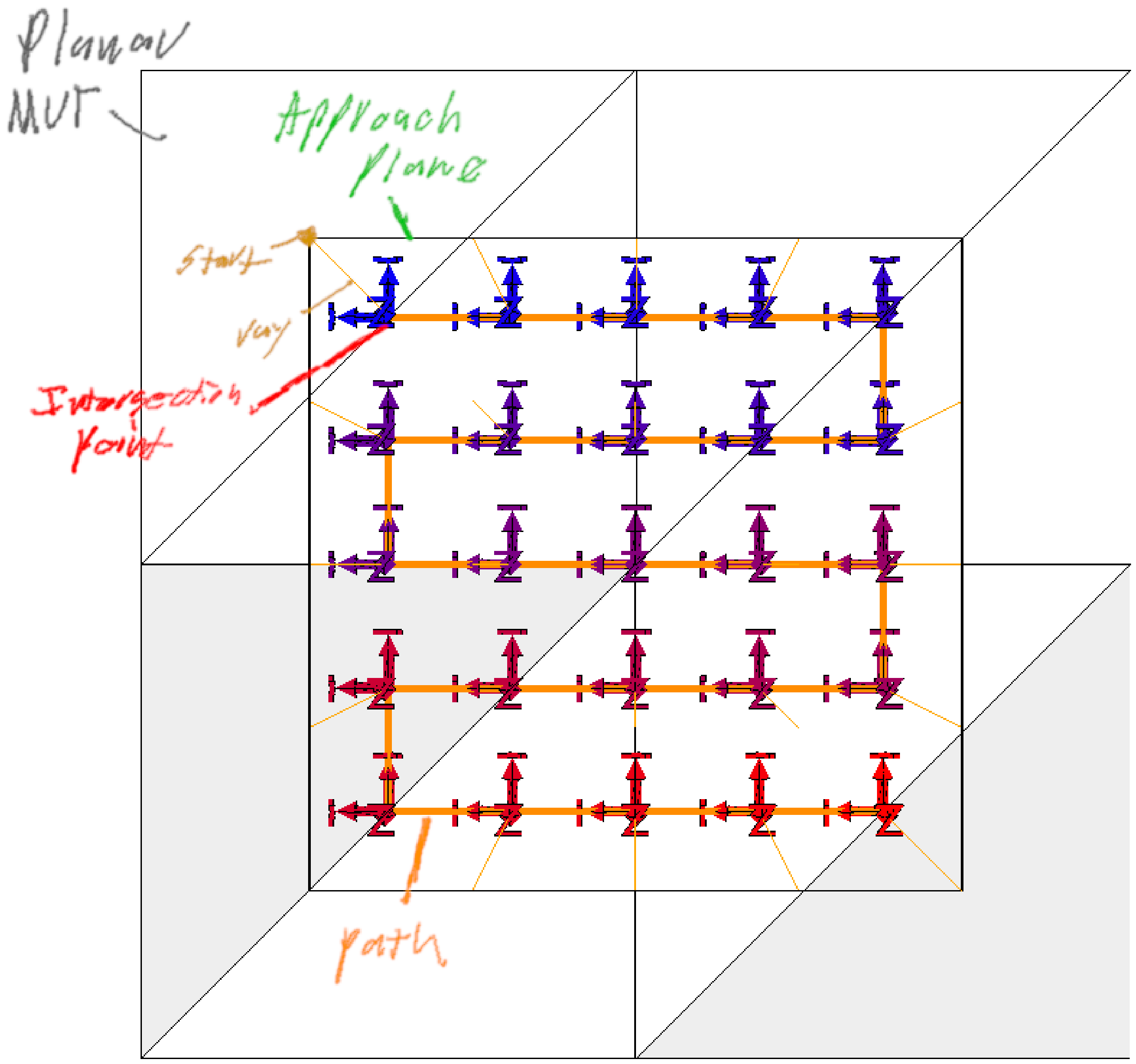


Figure 4.9: Plane-triangle example against a unit plane MUT with a bird's eye view

have little impact on the results for this paper since samples taken will mostly be smooth and there is a rare chance that a ray-edge intersection may occur while using a reconstructed image.

4.1.2.2 Organization Algorithms

Once the set of waypoints are known, then a toolpath can be organized from the known points. Such organization of waypoints is done through the famous traveling salesmen problem (TSP).

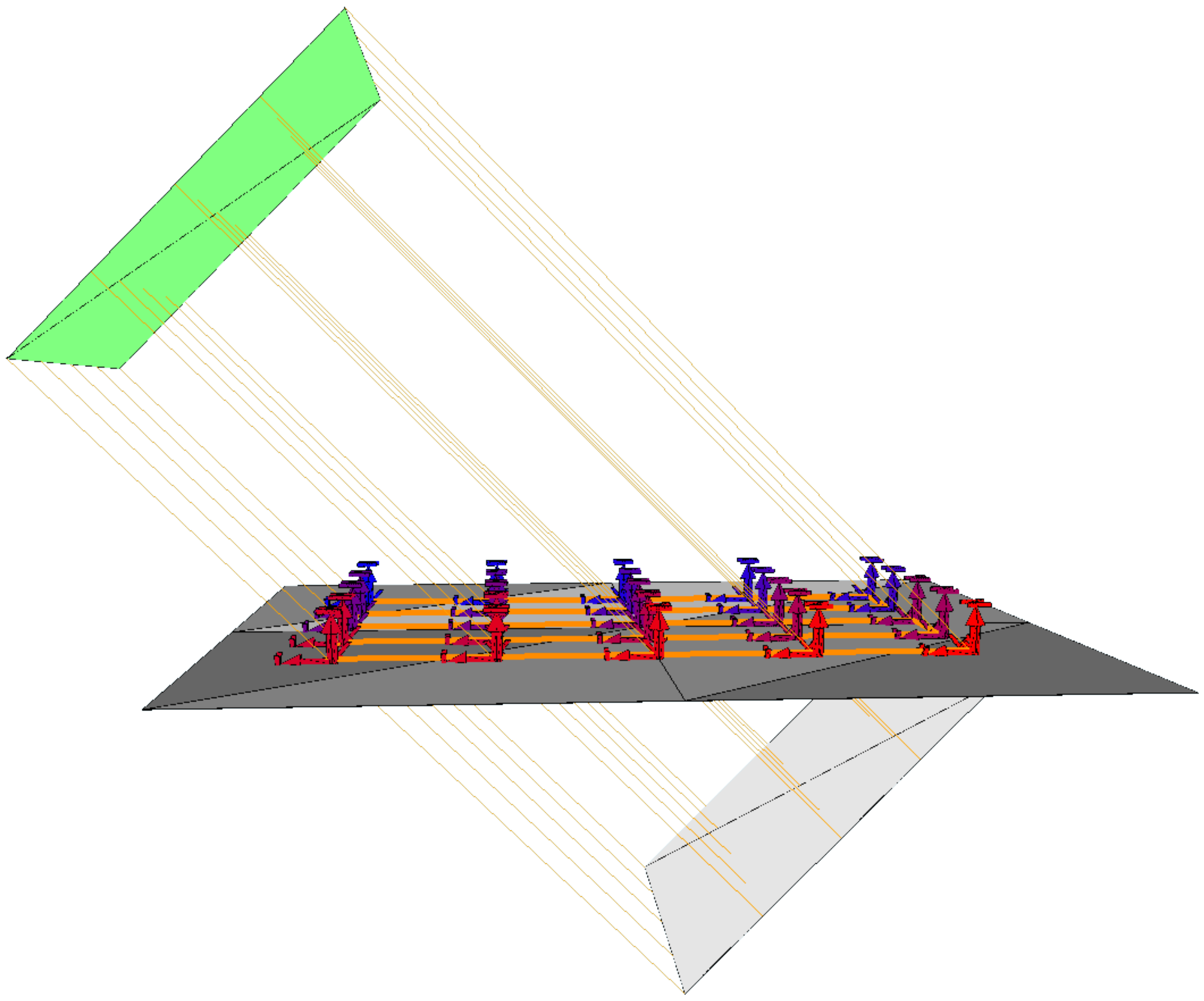


Figure 4.10: Plane-triangle example against a unit plane MUT showing the approach planes when the approach is tilted

The main consideration when designing TSP methods is to compare the amount of computation required to the time it takes to conduct a scan. It should also be considered the number of times the same scan is conducted within the same environment, for example if a similar scan is conducted across various samples. It might be worth conducting more time consuming method under these circumstances, especially since computation can be done passively once the physical environment is known.

There are several implementations of the TSP that weigh computation costs versus saving time, but implementation can be complex and time-consuming hence why only two are examined in

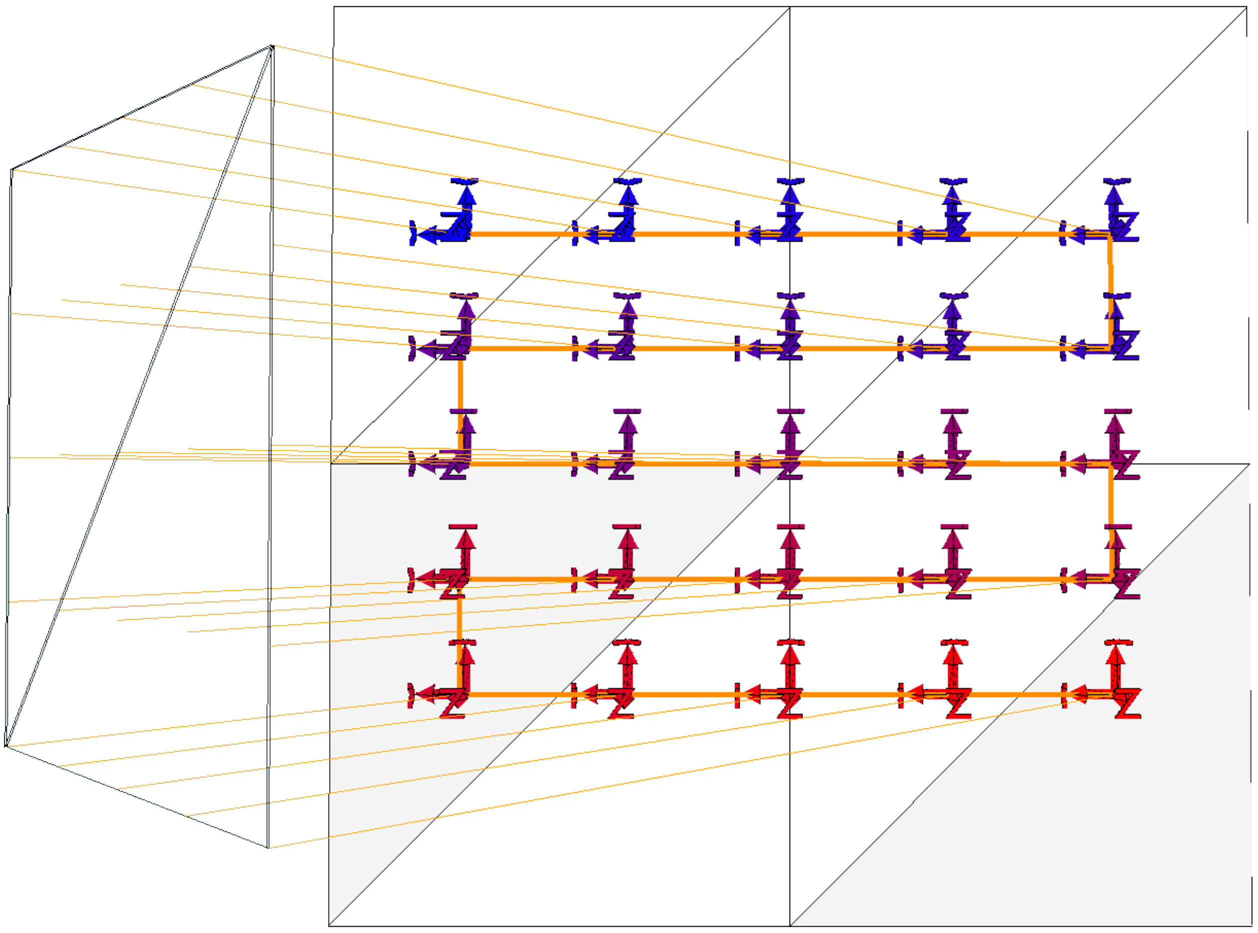


Figure 4.11: Plane-triangle example against a unit plane MUT with a bird's eye view when the approach is tilted

this report. To briefly describe the traveling salesmen problem, a list of waypoints is presented in which a "salesmen" needs to reach each point. There is a "weight" between each point, that can be described as distance, monetary cost, or amount of time required for reaching each waypoint. The goal is to find the most optimal solution to minimize the maximum weight of the entire "trip" without taking too much time deciding the best path. Accuracy or number of points that may be excluded are also factors, but are not considered in this explanation or system implementation. In the case for the actuation system, the best parameter as the weight is time, as time is relevant towards how many scans can be conducted and how fast results may be generated. However, from a computational stand point, precomputing the amount of time it takes from an actuation system is complex, requiring knowledge of higher order kinematic properties for each joint while presenting

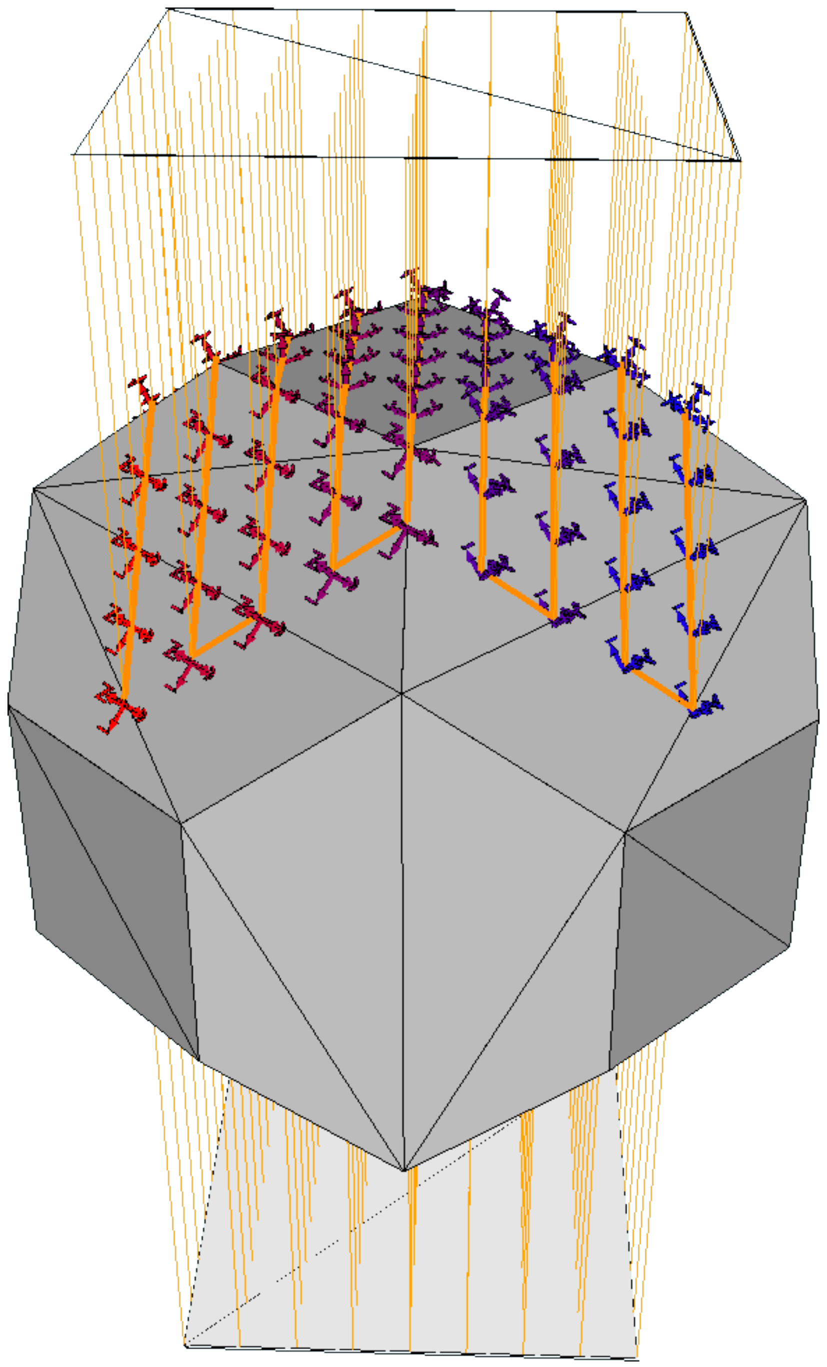


Figure 4.12: Plane-triangle example against a soccer ball MUT showing the approach planes

more overhead for TSP calculations. Since distance is a relative parameter for determining the amount of time, the distance between points is determined as the weight. Time, however, is the primary factor to be determined for weights and distance is only used as a massive simplification. In many cases, time should provide a better determination of weight cost of an entire scan, but most

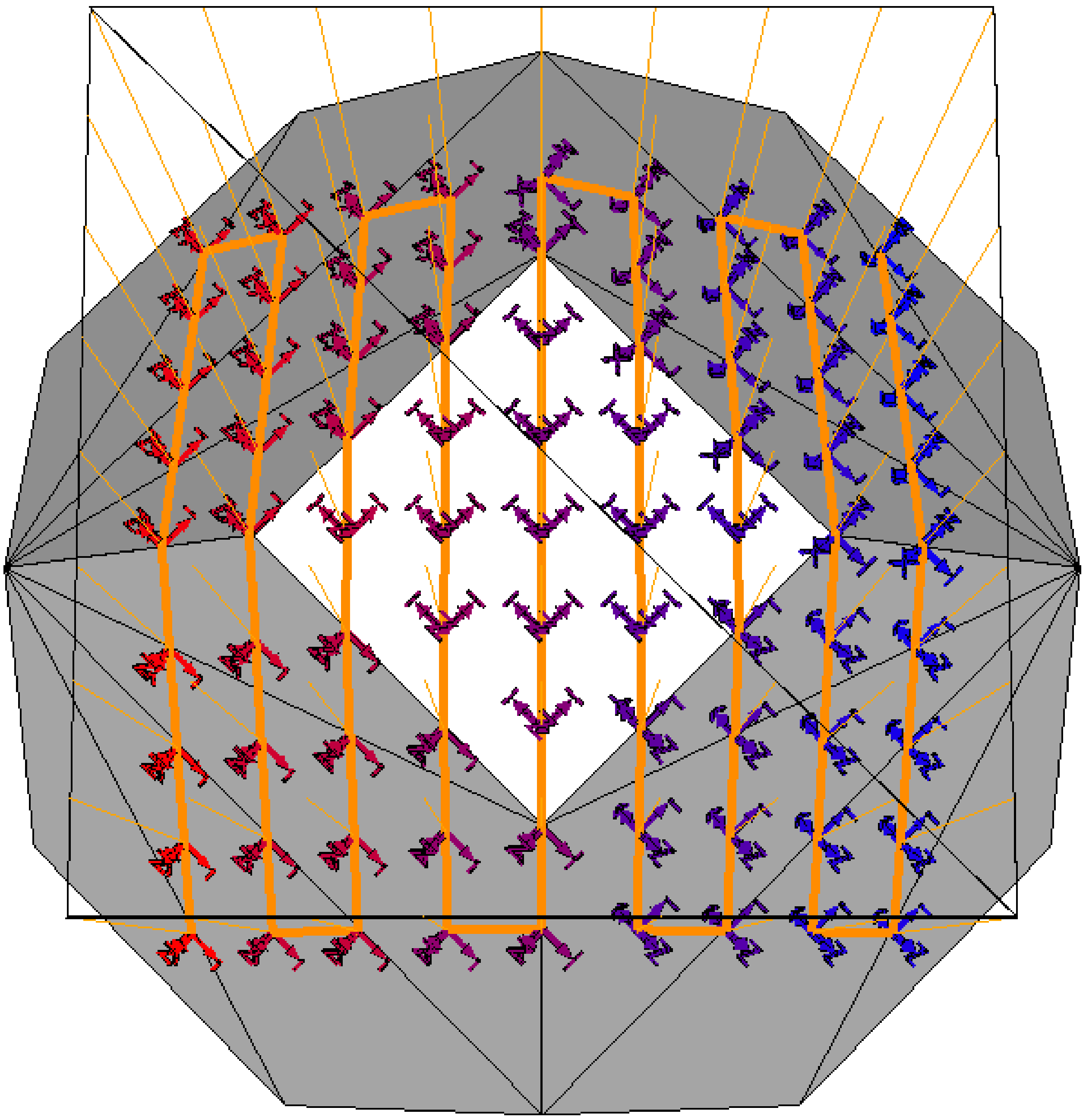


Figure 4.13: Plane-triangle example against a soccer ball MUT with a bird's eye view

likely not in computation time. Due to complexity, this is not examined but is highly considered for future implementation. The toolpath generator uses the zig-zag configuration a default if the zig-zag weight is lower than the TSP result. This is done as generating the zig-zag pattern is done before any TSP implementation, so the weight obtained for either the zig-zag or whichever TSP method can be compared, and if the zig-zag is lower, then it is used instead.

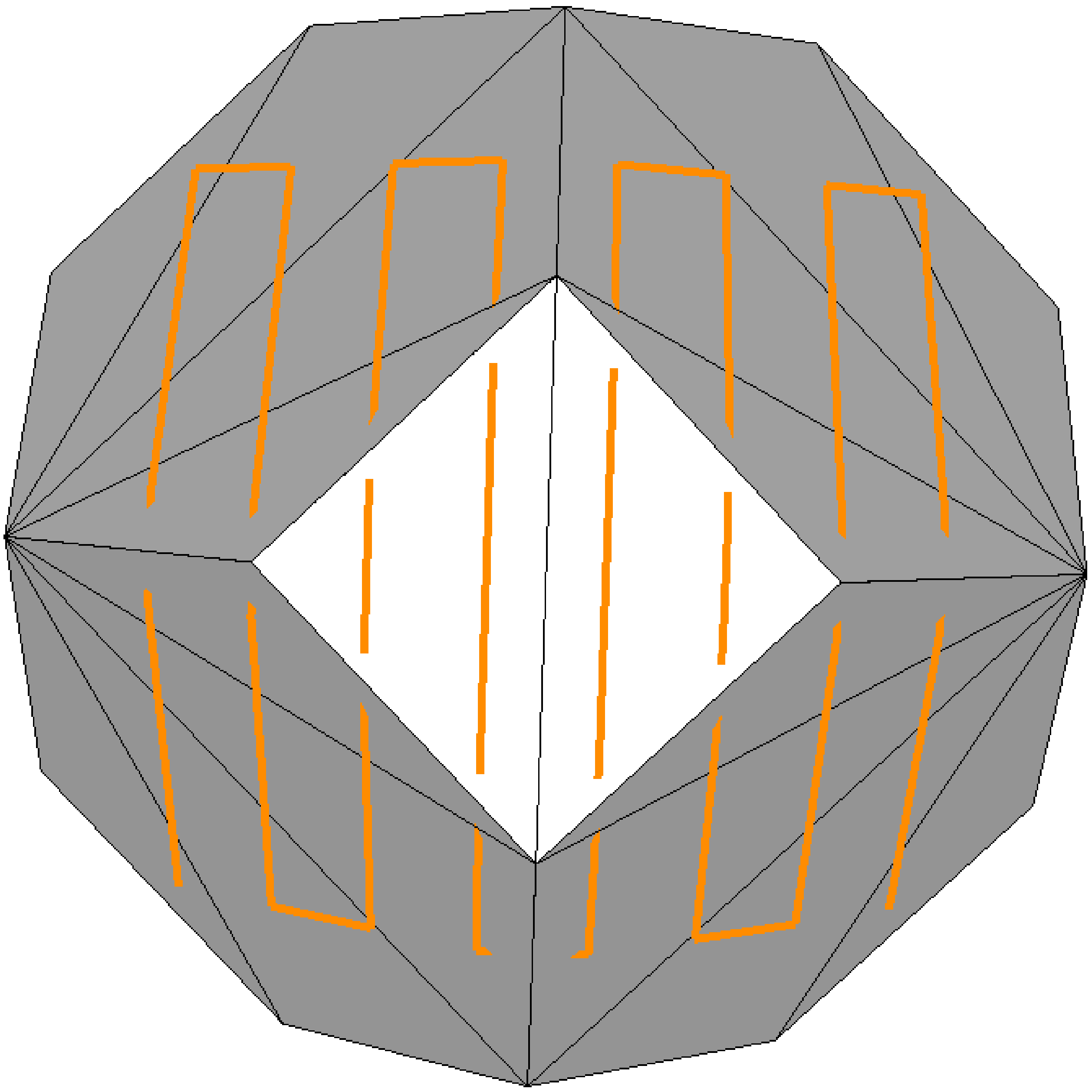


Figure 4.14: Plane-triangle example against a soccer ball MUT showing improper collision

Two TSP methods were examined for this report: greedy and exact. The greedy algorithm is a nearest neighbor approach to solve the TSP. The algorithm starts by comparing weights between the first point of the list with the rest of the list. The lowest weight is considered, and the first and second points in the toolpath are marked as "reached". From there, the second point finds the lowest weight to the third point, marking the third point as reached. This pattern of selecting the immediate best point is continued until eventually the entire waypoint set is covered, with the exclusion if a point is unreachable in which implementation is not in the program due to the complexity of collision detection. The nature of selecting immediately best path yields results that are typically better than the zig-zag pattern, while on average yields about $10-25\%$ less than the best possible (exact) path [4]. The main benefit of using greedy is that it's worst computation performance is $O(n^2)$ is very low in compared to other methods while obtaining a decently close approximation to the best path. The disadvantage is that it may generate randomness which creates crossings of paths already reached, and for some instances are biased towards having a larger weighted path if the final few points have large weights. In the case where the default zig-zag pattern has less total weight for traversing the toolpath then after greedy is employed, then the zig-zag path is selected instead. The greedy algorithm implementation for the waypoint organizer is shown in algorithm 4.3 which uses algorithm 4.4 to select the best point within the list. Algorithm 4.5 would be used to select the type of weight, such as distance or time. Time calculation is not implemented due to knowledge of actuation system components which are handled in separate software and would require integrate.

The exact method determines the path which yields the least weight out of all paths in the list by employing brute-force and checking every combination available. Implemented for this report, the exact algorithm used a recursion loop that would keep track of each weight at each point, and append for each combination. While looping, each point that is entered into the list is marked as "reached" to avoid redundant calls while appending the local weight within the function. The recursion loop reaches the final recursive call when the number of waypoints in the list, in which the total weight is checked with the relative current best weight, which starts as the zig-zag toolpath weight. If the current path's total weight, then the current path and total weight is marked as the current best. If a the weight of a tested path exceeds the overall best weight then it is skipped. While yielding the absolute best path, computation time for this method can be absurdly high, with the worst computation performance being at $O(n!)$. Effective matters to decrease the amount of computation per step can help decrease the computation time significantly. To avoid redundant calculations, creating a $2D$ array of weights between each point before applying the

Algorithm 4.3: Greedy TSP

Input: The enumeration value of which weight mode to use, $WeightMode$, a list of waypoints, $InputPoints$

Output: A list of waypoints organized using the greedy algorithm, $OutputPoints$, the non-negative value for total weight of movement, $TotalWeight$

```
1: $length = InputPoints.Count$;    ▷ The number of points inside of the input points list
2: $OutputPoints = new\ PointsList(length)$;    ▷ Set up the output path as an empty point list
3:     ▷ with the capacity at $length$
4: $currentPoint = InputPoints[0]$;    ▷ Select the current point as the starting point
5: $currentPoint.Reached = true$;    ▷ Set as reached by default
6: $TotalWeight = 0$;    ▷ Set up the total weight traversed as zero
7: for $i = 1; i < length; i = i + 1$ do    ▷ Iterate through all points after first, solve the best weight
8:     $weight = 0$;    ▷ The local weight solved from the previous
9:     $point = BestGreedyPoint(InputPoints, currentPoint, out\ weight)$;    ▷ Solve
10:    if weight $\neq 0$ then    ▷ If the weight is solved (it's not equal to zero)
11: ‘    ▷ then a valid point was found. Do the following:
12:        $point.Reached = true$;    ▷ Mark the current point as reached
13:        ▷ Note: by default, $points.Reached = false$
14:        $OutputPoints.Add(point)$;    ▷ Add the point to the path
15:        $currentPoint = point$;    ▷ Set the current point as the point selected
16:        $TotalWeight = TotalWeight + weight$;    ▷ Append the local weight to total weight
17:    else
18:        $break$;    ▷ If no point found, then leave the loop to exit the function
19:    end if
20:    $OutputPoints.ResetReach()$;    ▷ On the final list, restore default reached configuration
21:    $return\ OutputList$;
22: end for
```

exact TSP algorithm, and calling the address of the index of the weight between points while the algorithm were to run. While this is assumed to save much time, if the number of points exceeded around 20, then finishing the algorithm would take an extreme amount of time to compute with $20! = 2.432902 * 10^{18}$ instances to compute. Therefore, this method is decent if there are a low number of points, but unusable if a large number of points are available. It is better approach is to increase the optimization use dynamic programming methods to optimize the exact TSP solution, which methods exist for bi-directional weight configurations [14]. The implementation for exact TSP is used within the software, but is not used further use to incredible waiting times for even small-sized waypoint sets.

Algorithm 4.4: Best point solver within the Greedy TSP

Input: The enumeration value of which weight mode to use, *WeightMode*, a list of waypoints, *InputPoints*, the current point to compare with the rest of the list, *CurrentPoint*
Output: The best point solved (if not solved, this it is *null*), *BestPoint*, the non-negative value for best weight solved (if not solved, then is equal to positive infinity), *BestWeight*

```
 1: $BestWeight = \infty$;                         ▷ Set the best weight to positive infinity
 2:                                             ▷ as any other value selected will be better
 3: $BestPoint = null$;        ▷ Set the best point to null, which is the default if no point was found
 4: $length = InputPoints.Count$;                 ▷ The number of points in the input point list
 5: for $j = 1; j < length; j = j + 1$ do                          ▷ Iterate through the list of points
 6:     $point = points[j]$;                  ▷ Get the local point with the index of this loop
 7:     if $localpoint.Reached == true$ then
 8:         $continue$;        ▷ If the local point has been reached, immediately go to the next point
 9:     end if
10:     $weight = GetWeight(CurrentPoint, point)$;                    ▷ Compare the weight of the
11:                   ▷ input point (CurrentPoint) and the local point, and obtain the weight value
12:     if $weight < BestWeight$ then       ▷ If the local weight is less than the current best weight
13:                                           ▷ then set the best parameters as the local ones
14:         $BestWeight = weight$;
15:         $BestPoint = point$;
16:     end if
17:     if $weight = 0$ then
18:         $break$;                    ▷ In the case that the weight is zero, then the current point and
19:                   ▷ local point are the same. In this case, then leave the loop to exit the function.
20:     end if
21: end for
```

Algorithm 4.5: Best weight selector

Input: The enumeration value (of type *WeightModes*) of which weight mode to use, *WeightMode*, and the two waypoints to compare, *WaypointA* and *WaypointB*
Output: The weight between the two points, *Weight*

```
1: switch WeightMode do        ▷ Switch based on the weight mode, either being distance or time
2:     case $WeightModes.Distance$                    ▷ Simply, the distance between points using
3:                                                         ▷ translational $T_{xyz}$ parameters
4:         $return\ WaypointA.DistanceBetween(WaypointB)$;
5:     case $WeightModes.Time$                                  ▷ Time is to be implemented
6:                                            ▷ would be a separate function due to complexity
7:         $return\ SolveTime(WaypointA, WaypointB)$
```

4.1.3 SprutCAM Toolpath Generation

For the toolpath generation, SprutCAM 12 was used due to it's powerful set of tools and morphing functions required for accurate toolpath computations. While a wide range of solutions are possible even only considering the numerous amounts job types and strategies available, it was selected to use the $3D$ and $5D$ surfacing operations through mostly automatic means. This configuration allows for a raster-like movement around CAD-defined regions of interest over flat or curved surfaces while keeping the end-effector normal to the surface for each movement. It is also possible to scale the NDE scan resolution by adjusting the end-effector tool's diameter. The output of SprutCAM's actuation simulation will be an array of movements in S. Because of the customization of the ray-triangle intersection path-planner allowing for fine-tuning of what is needed for NDE scans, and in this sense providing potential fixes to limitations of SprutCAM, there is more discussion and experimental results of the custom toolpath planner than SprutCAM. This is also considering that most usage of SprutCAM is already well documented and including it here would be redundant That being said, SprutCAM is a very powerful software that is considered for future projects but is less appropriate to use for the purpose of this thesis. Since SprutCAM was useful for a portion of the research, it will still be briefed.

There are some instances where the end-effector will not be along the sample, which will be considered as the probe being in the air. Two significant instances that will occur during each scan will be when the probe moves from the starting home position to the first point on the samples, and when the probe moves from the last surface point on the MUT back to the home. Other instances occur due to scan path optimization where the probe may jump in the air, possibly each across the sample itself. These points can be filtered out either by interpreting the commands outputs from SprutCAM, which would be considered extra information included with the S output, or by filtering out these points via post processing. Figure 4.15 shows a simulation within Sprutcam using a Fanuc ARCMate 100iB robot arm as the actuation system and an x-brace as the MUT. Note that the MUT is predefined so the faces would can be selected for scanning, as it was defined by a STEP (Standard for the Exchange of Product model data) file rather than as a mesh. The scan type used

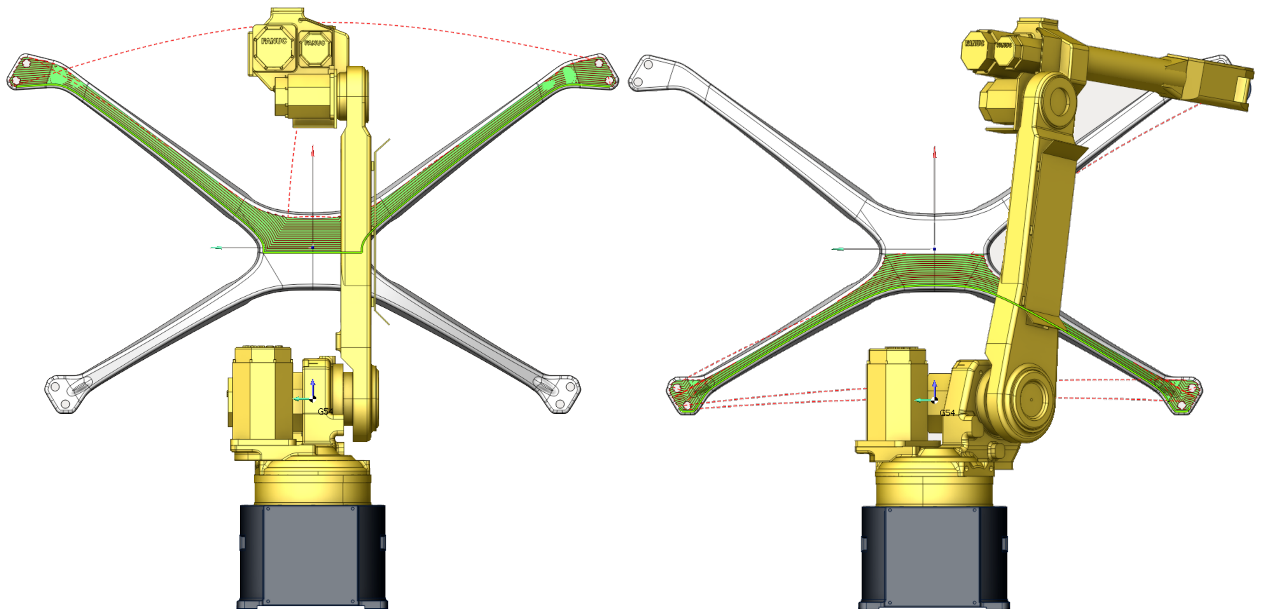


Figure 4.15: SprutCAM path planning using an x-brace sample partitioned twice

was $5D$ surfacing which allows for raster movements along user-chosen faces and curve to move along from, definable by the edges of the faces used.

Some other benefits for SprutCAM is generating a path which avoids collisions with the MUT and the robot itself. It should also provide collision avoidance for obstacles within the workspace, such as the specimen holder. There are a couple of limitations where SprutCAM or any other toolpath generator cannot solve due to either the complexity of the MUT or limitations of the actuation system. This may occur if the specimen contains points where there is no possible path for the robot to reach, such as a concave pattern too small or complex for the robot to actuate or if the specimen is too large to be within the bounds of the workspace. If the model has complexities which might invalidate the scan or lead to optimization issues, it is possible to partition regions to scan. This is shown in figure 4.15, where the top is scanned first then the bottom.

There are two possible operations in SprutCAM that can provide $5D$ surfacing results normal to the surface of the virtual MUT. First is the $5D$ surfacing which requires a CAD-defined model. For this, the mesh model used from reconstruction needs to be converted to a CAD file such as STEP. which requires the input of which faces will be scanned as well as start and finish curves which are selected from the edges of the CAD-defined model. There are multiple strategies which are possible and yield various results, but the morph between curves strategy was mostly examined. The second operation is $5D$ morphing, which allows a user-defined region where a surface scan would take place within a $2D$ boundary, including two start and stop splines with two sync lines. The main difference of this operation is that a select region on the sample may be selected for scan path region that is not just based on the faces on the model, therefore regions inside of faces may be scanned. The $5D$ morphing operation does not require using a CAD-defined model, though it might be more advantageous for one since the edges can be selected for the region to select. There are many possibilities that SprutCAM provides for path planning that have yet to be tapped, and considering that it's a commonly updated software, there might be future improvements that make it beneficial for usage.

4.2 Interfacing Actuation and NDE Systems

The following sections are not covered within the experimental results, but are here for future implementation. Most of these systems have been developed alongside reconstruction and path planning features, but were not developed enough to be covered in this report. This is in part to robot system complications and required validation of physical data. Once the toolpath is generated, then the commands from the tool are to be sent towards the actuation system. The system requires programming based control to synchronize the toolpath position with the NDE data stream. To allow for this, RoboDK was used. This software uses the list of forward kinematic transforms S provided by SprutCAM as input to simulate the scan real-time while connected to a C# software dealing with NDE data acquisition. RoboDK allows for running functions from C# externally, as well as Python internally, which is used to synchronize the actuation system movements within both virtual and physical space with the data output of the probe. As such, useful interfacing measures such as collision detection are also implemented. Synchronizing these two systems allows for combining the the S data of the probe as the end-effector towards the probe's output. In other words, the data output will acknowledge the NDE information with respect to the workspace environment.

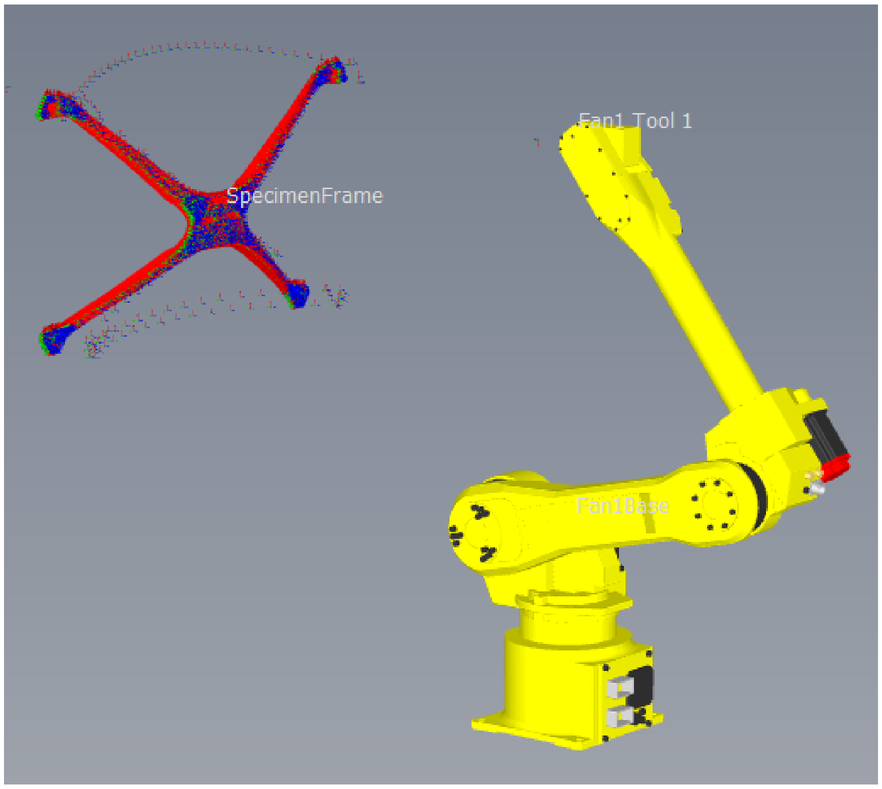


Figure 4.16: Robot setup scanning of an x-brace sample within RoboDK

As the scan moves along, an array of non-uniform vertices are generated creating a point cloud of NDE data. This NDEpoint cloud should not be confused with the point cloud of the MUT used for the surface reconstruction of the toolpath. Colorization of the point cloud can be determined by combining the output of the probe to one value per point, and scaling each point's NDE value within a defined color map between minimum and maximum value of the NDE probe's output to obtain a color value $P_{C_{rgb}}$. The default NDE values found during the scan should also be saved for other post processing means, however large arrays that are generated such as within ultrasonic scanning may be needed consideration due to memory and computation constraints. It is possible to scan each data value only at each defined toolpath point, or to continuously obtain points regardless the current pose and interpolate the position of NDE data points alongside the defined toolpath.

The resultant of this system will be the image I_{nde} which holds the information needed for post processing. The real-time output of these results will be a colorized point cloud where new NDE data points are appended among other vertices of the scan. When scanning is finish, post processing methods may be done to further clean and possibly reconstruct the image into a mesh. The direct benefit of this would be to be able to obtain edge and face information of the data point cloud. In turn this would add interpolation of the points within space allowing for easier visualization of the model and more options for data interpretation of damages or irregularities. These methods would be very similar to how the MUT was reconstructed earlier, where the image is cleaned of any outlier points such as when the probe is in the air, the Poisson filter reconstruction process, and possible mesh smoothing or other enhancement methods. Since this requires a working physical system to obtain data, this process is saved for a future report.

4.3 Physical NDE and Actuation System

The physical system will to be used for experimentation will be briefly explained as this report focuses mostly on the virtual implementation of the system. The output of this system should be the array or image of NDEpoints I_{nde} which will be processed and visualized later. As the physical system is unavailable, the data shown in this section is more for theoretical purposes, and as such will not be shown in the experimental results chapter. An air-coupled ultrasonic system was developed to work alongside the virtual system, and was used for sending and receiving the signal with minimal loss while allowing for multi-probe sensing with four channels. A Sonoair system from Sonotec was used. The output signal of the Sonoair system goes into a Gage card placed on the motherboard of the CPS controlling computer, which is visualized, interpreted, and saved at real-time. In order to allow the pitch-catch nature of the air-coupled setup, a dual robot system was designed and the flow of the system is shown in figure 4.17. The challenge with this setup is that the robots must be very close to normal against each other at all times, or else major gain losses are introduced. In RoboDK, it is possible to synchronize robotic kinematic movements by calculating the velocity in which both robots move for each point within the input toolpath.

The output of the physical system will be a set of data that may be visualized either using a point cloud viewer software, such as Meshlab or CloudCompare. Using OpenTK, the output of a simulated real-time scan is shown in figure 4.19, where each data point only holds it's transformation

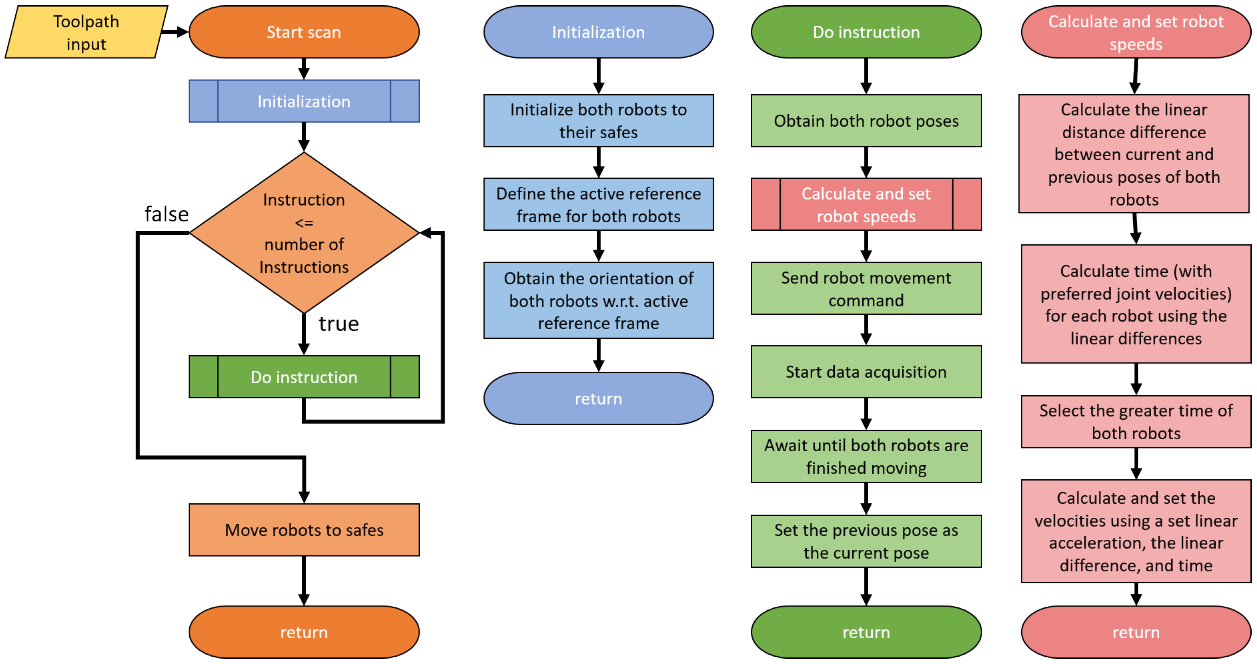

Toolpath input
Start scan
Initialization
Instruction <= number of Instructions
false
true
Do instruction
Move robots to safes
return
Initialization
Initialize both robots to their safes
Define the active reference frame for both robots
Obtain the orientation of both robots w.r.t. active reference frame
return
Do instruction
Obtain both robot poses
Calculate and set robot speeds
Send robot movement command
Start data acquisition
Await until both robots are finished moving
Set the previous pose as the current pose
return
Calculate and set robot speeds
Calculate the linear distance difference between current and previous poses of both robots
Calculate time (with preferred joint velocities) for each robot using the linear differences
Select the greater time of both robots
Calculate and set the velocities using a set linear acceleration, the linear difference, and time
return

Figure 4.17: Synchronization setup of a dual robotic NDE system

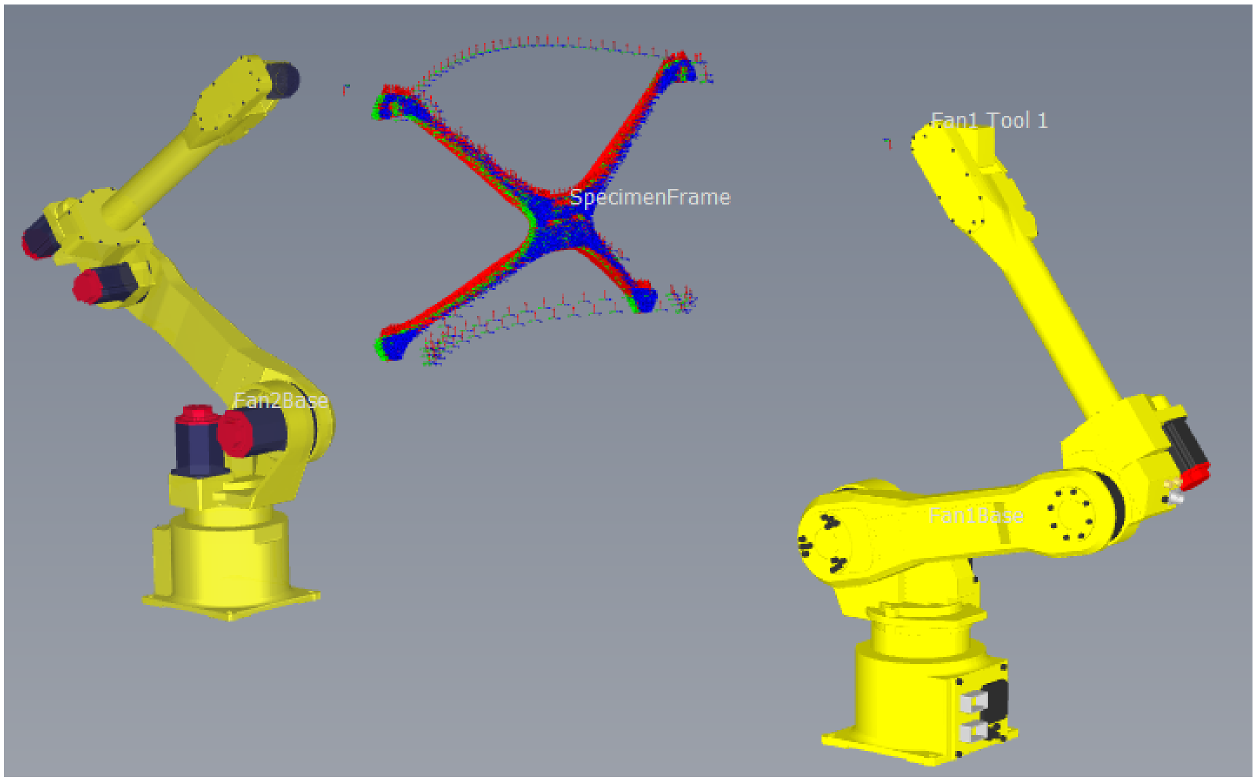


Figure 4.18: Dual robot setup for scanning an x-brace within RoboDK

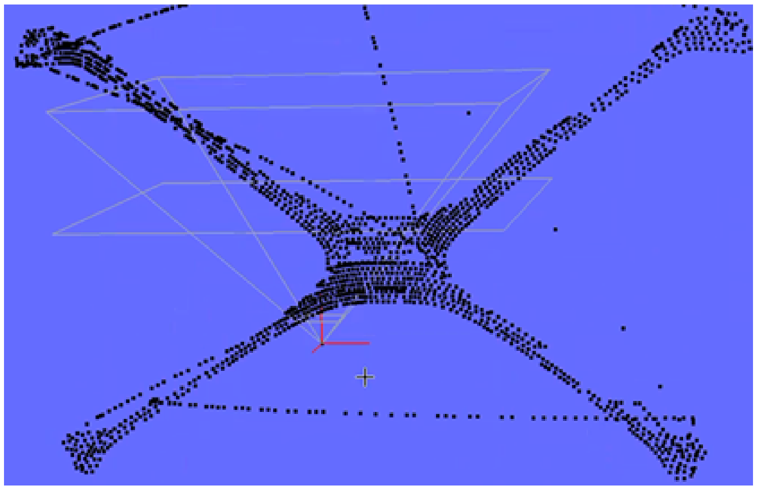


Figure 4.19: Simulated real-time data of the x-brace sample from RoboDK, visualized using an OpenTK visualization software

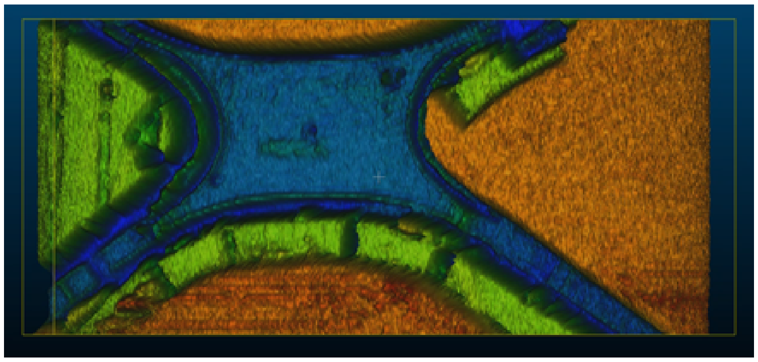


Figure 4.20: Point cloud visualization of $2D$ ultrasonic data from a x-brace sample using CloudCompare

parameters p with respect to the virtual environment. As this is a virtual example, the nde information $Block_{nde}$ is not available, hence why each point is defaulted to black. Colorization is possible when NDE data is available, where a color normalization map N_{map} can be generated from minimum and maximum output NDE data. An example using $Block_{nde}$ and N_{map} is shown in figure 4.20 using data obtained from a gantry running a planar raster scan over an x-brace. The combination of these two concepts should yield data from the $5D$ scan which may be reconstructed into a $3D$ mesh with normal information using similar processes as the MUT reconstruction.

CHAPTER 5

EXPERIMENTAL SETUP AND RESULTS

For this section, there were several experiments that have been conducted which follows the steps of reconstruction, toolpath generation, and simulated movements discussed in the previous sections. The first experiments will using CAD-defined models which will show basic reconstruction steps and comparisons with the CAD-defined models. Following these experiments, reconstructed models from a Intel RealSense d435i camera was used. This camera was selected, as obtains high quality depth image results with easy library interfacing with the *C#* software. It is also equipped with a gyroscope and accelerometer, however these were not used due to inaccurate results. Two sets of tests were conducted: camera-as-end-effector tests, where the camera was mounted on the end-effector of the Fanuc 100iB robot arm, and one set where the camera was positioned multiple times on a flat table for multi image reconstruction. For the camera-as-end-effector tests, three sets were obtained. One was a box test, where three boxes were stacked on top each-other which allows for detecting where the end-effector lays while providing low-risks for damaging the camera or samples. The next sets used three carbon-fiber car samples, with one set placed on a flatbed or table and scanned overhead, and the other with the car pieces placed vertically on a vice.

To avoid redundancy, the descriptions of each sample used from the stereo camera are explained here. For the box test, each box is labeled as "box1", "box2", and "box3" where box1 is the top box, box2 is the middle box, and box3 is the bottom box as seen in figure 5.1. The dimensions, where length is from left to right, height from top to bottom, and depth facing away from the viewer, are $5 \times 12 \times 10$ inches for box1, $10 \times 16 \times 10.5$ inches for box2, and $10 \times 14 \times 12.75$ inches for box3. Note that obtained images do not cover the entirety of box3, so it's height is not fully shown. For the car samples, each is labeled as "car1", "car2", and "car3". Figures of cars 1-3 are shown in figures 5.2, 5.3, and 5.4, where each is placed on the vice. Each car piece has variations in depth, so approximate dimensions are given. The pieces car1 and car2 contain an internal honeycomb structure, which have depth variances for locations with and without this honeycomb structure.

For the car1 piece, there is a curved ramp between these sections, where the portion without the honeycomb structure is on the bottom right as seen in figure 5.2. Approximately, the sample is 14×9, with the top part measured approximately at 8 inches and the honeycomb structure difference being around 0.75 inches. For car2, the honeycomb structure is to the right of the sample as seen in figure 5.3. There is also a major ramp change as seen in the left of the sample. The dimensions were approximately 21×7, with about 3 inches for the main ramp on the left, and approximately 0.75 inches for the honeycomb dip. For car3, no honeycomb structure is present so thickness is mostly the same. However, there are many extrusions on this sample. Partitioning the piece into the top left section as seen in figure 5.4 and the bottom section, the top portion was measured at around 6×6 inches, and $27x9$ inches for the bottom part. With various extrusion components, there is a difference in depth around 2 inches throughout the sample.

There are some improvements for future results. First, each of the car images from the camera-as-end-effector were taken from single shot images. More information around the samples would be advantageous, but due to robot setup complications location was difficult to validate. As such, location of the samples with respect to the robot were not fully validated to ensure proper robot movements with the samples, which is important to fix for later usage. The sample car1 is used for multiple translation testing to show the center point transformation technique, but rotation complicated this process due to some unknown distortion with the point cloud images obtained. Other data sets were obtained, such as a recorded session around complex shapes where ICP is used, but not shown in this report due to logistical reasons. Also, there are optical illusion issues that occur. Shadows from the coloration of the mesh may make the mesh seem to change in depth for instance, but the mesh might still be flat. It is possible than an "RGB stretch" occurs, as the region of the depth sensor is larger than the RGB camera, RGB values for point clouds are extended for missing points. As such, it's better to use depth images. Finally, some point cloud images in the path planner viewer were shot from the back rather than front, due to 3D engine issues showing the path over the sample. This relates to the issue shown in figure 4.14, where the path "enters" the mesh. Any minor placement of the ray behind the mesh will still show it behind the mesh. The meshes were designed to be smooth, so collision issues are minor.

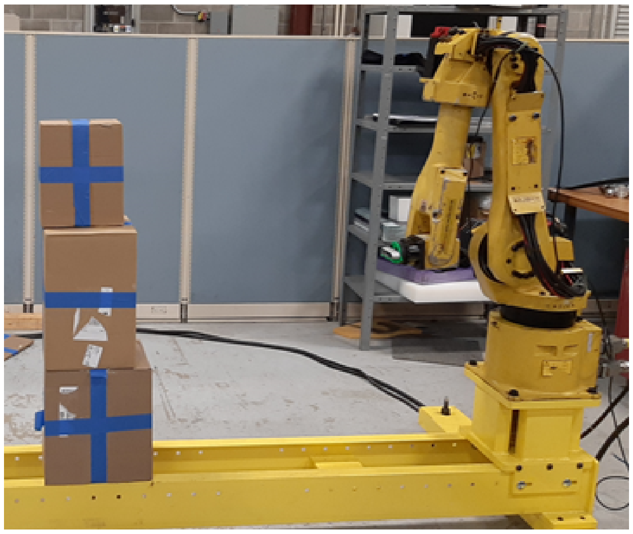


Figure 5.1: Physical setup for the box test, camera at position 2

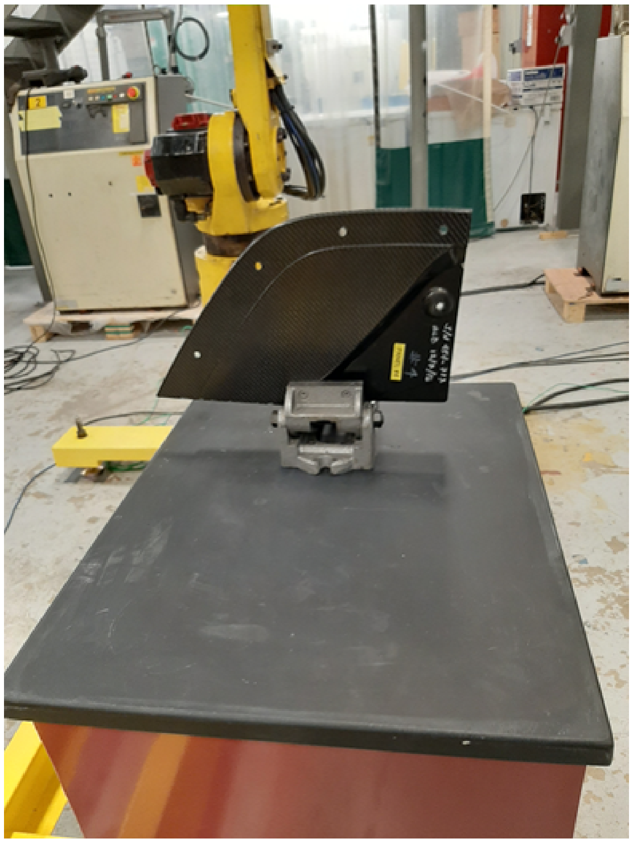


Figure 5.2: Sample car1

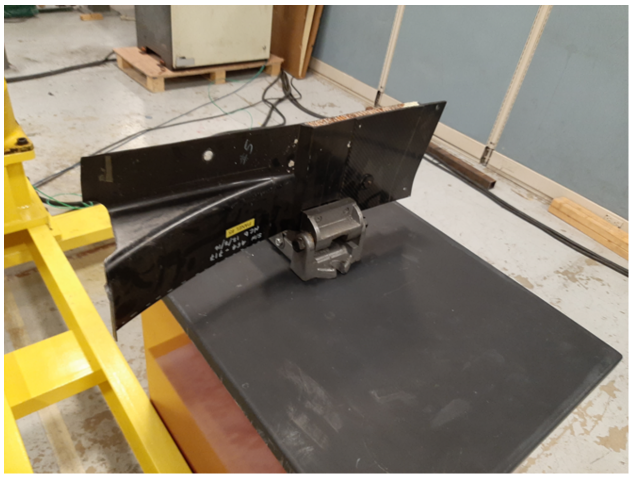


Figure 5.3: Sample car2

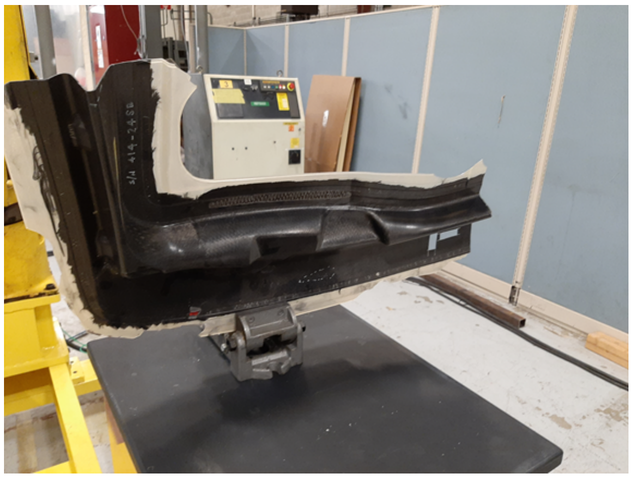


Figure 5.4: Sample car3

5.1 Simulated Flat Panel

The first experiment covered is the flat sample scan which is used to simulate a panel-like MUT within the scanning environment. Both a full-region and sub-region scans were conducted, where the sub-region covers a quarter of the panel at the center. Generally these samples can be relatively easily scanned using a gantry and running a $2D$ raster scan, however the purpose is show the same action using the robot arm setup due to it's simplicity. The panel is defined to have a $2m$ width and height. The original model only has one face with four vertices, which is subdivided four times to have 17 by 17 points across the model. Poisson reconstruction was used at depth level 8, as seen in figure 5.5, and then went through a Laplacian filter with ten iterations, as seen in figure 5.6. The white dots are the vertices which guides as an example of what the flat panel should look like. In the reconstruction process, some of the information from the edges were removed due to the rounding. Note that the vertices of the original model were not deleted in the front view images and that they are in the back of the model due to surface curving. The reconstructed model is then placed into SprutCAM for toolpath generation, along with the original flat model for reference. For the full region scan, the $5D$ surface operation was used while the The $5D$ morph operation was used for the sub-region. The sub-region is shown just to show as a possibility for scans, where multiple segments can be partitioned to possibly save scan times. Both scans were generated using $200mm$ step-sizes, where 11 main scan steps along side with the width $2m$ will have the path be set. Ideally, the traversed distance will be $(11 * 2m)m + 11m = 24m$. Figure 5.7 shows both $5D$ surfacing and morphing operations for the reference panel. The reconstructed piece had difficulty being used in SprutCAM, including measuring the total distance and difficulties generating proper paths with reconstructed models. The custom path-planner will be used instead from this point forward because of this issue.

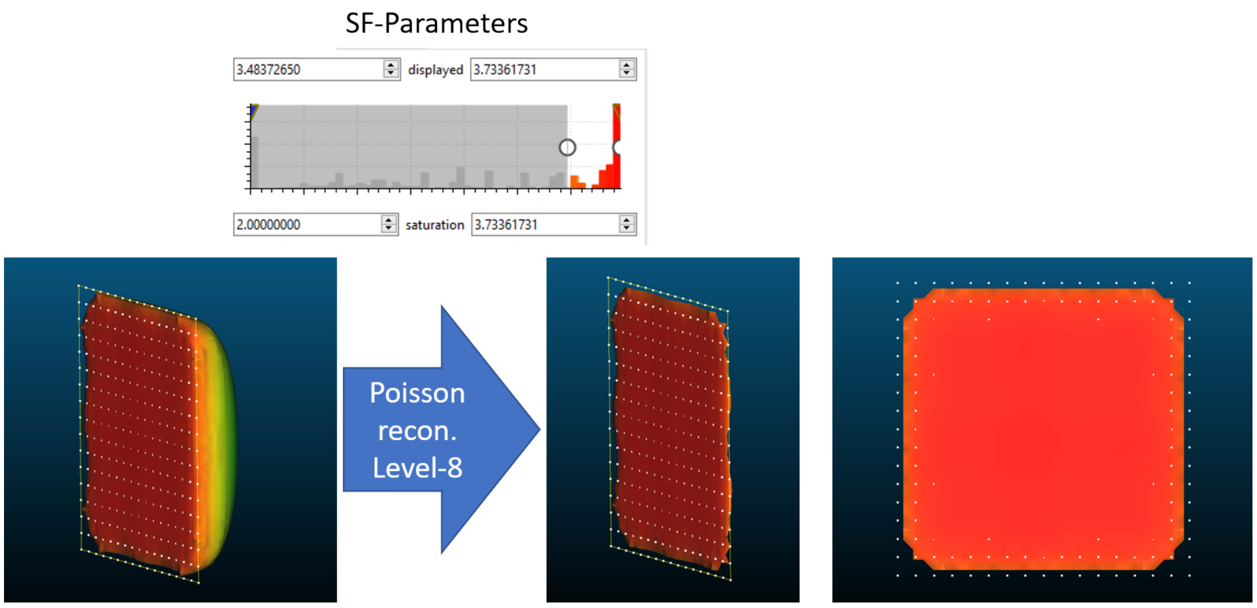


Figure 5.5: Flat panel reconstruction in CloudCompare

The flat piece with and without reconstruction were both examines, as shown in figures 5.8

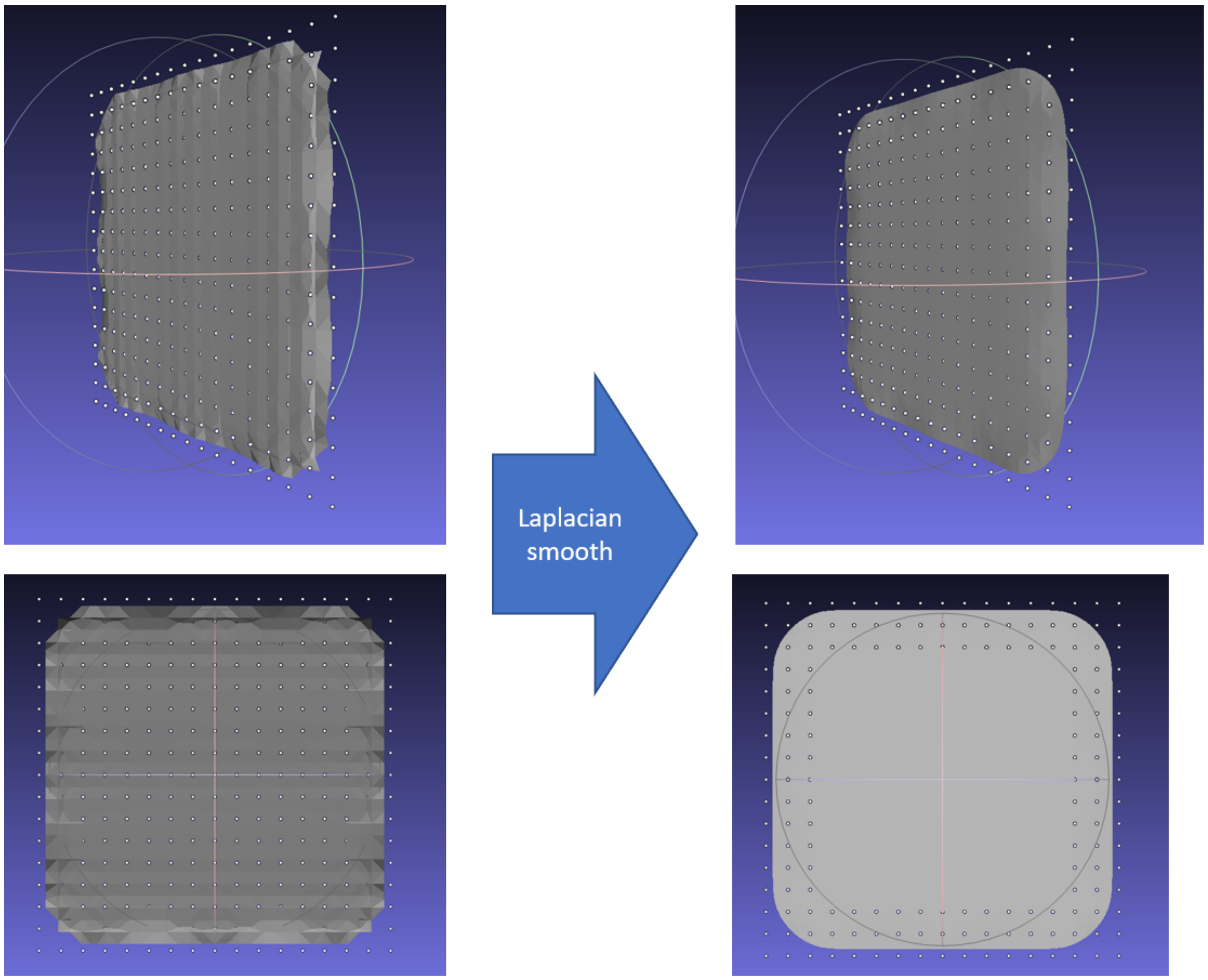


Figure 5.6: Laplacian smoothing on the reconstructed model in MeshLab

and 5.9. Only the zig-zag pattern was used for this experiment, as the greedy algorithm produced similar results. This is due to a selection process where in which if the results for greedy yielded poorer results than the zig-zag pattern, then the zig-zag is used instead. The same step-sizes were used as for the full scan. As recalled in a previous section, the ray plane is used to generate intersections alongside the mesh in order to parse each normal on the surface of the MUT, resulting in waypoints alongside the mesh used for path planning. For the reference panel, the ray resolution of 11×30 was used with the ray plane scale was set to $2 \times 2m$. The ray plane approach was set to be normal to the surface of the sample. The resulting path was measured as $23.713028m$, where some space was lost due to a minor bug where the next path starts slightly ahead. This distance has

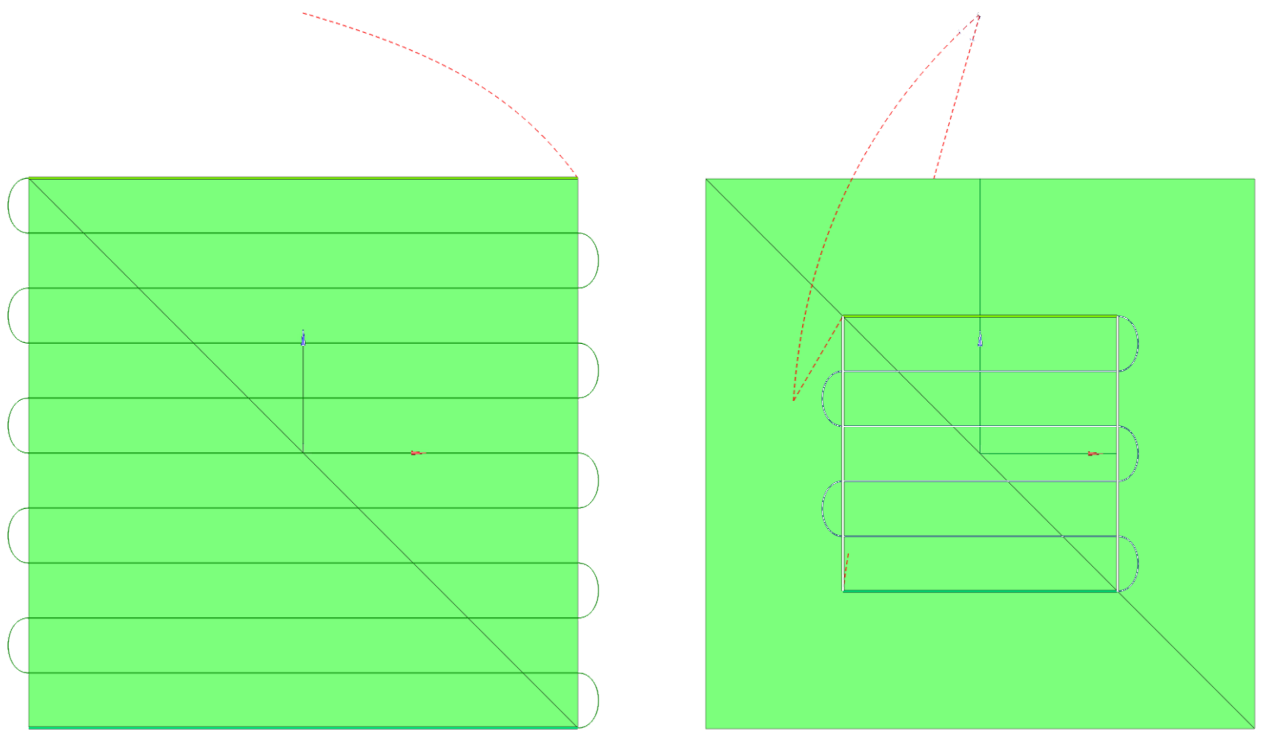


Figure 5.7: 2m square flat reference panel full-region (left) and sub-region (right) toolpath generation in SprutCAM

a percent error of 1.19572% from the ideal distance. For the reconstructed plane, the same 11×30 resolution was used and approach, but since the model is scaled slightly less than the reference, the ray plane was scaled to $1.9 \times 1.9m$ instead, where the ideal traversal distance would be $22.8m$. The distance traversed was $21.696026m$ with a percent error of 4.85954% with $22.8m$ traversal distance. A minor amount of distance is also added due to slight changes in depth.

5.2 Camera-as-End-Effector Tests

The purpose of the experiment was to obtain a general reconstruction of multiple images along the car piece and then obtain a toolpath within a certain region using partitioned sub-regions from the custom path planner. The following tests were conducted using the camera as the end-effector of a Fanuc ARC Mate 100ib robot arm, shown in figure 5.10. The advantages of this setup is that the camera can be transformed anywhere within the arm's workspace while remaining steady for still shot images. Hence why it was used for a large portion of the reconstruction and path

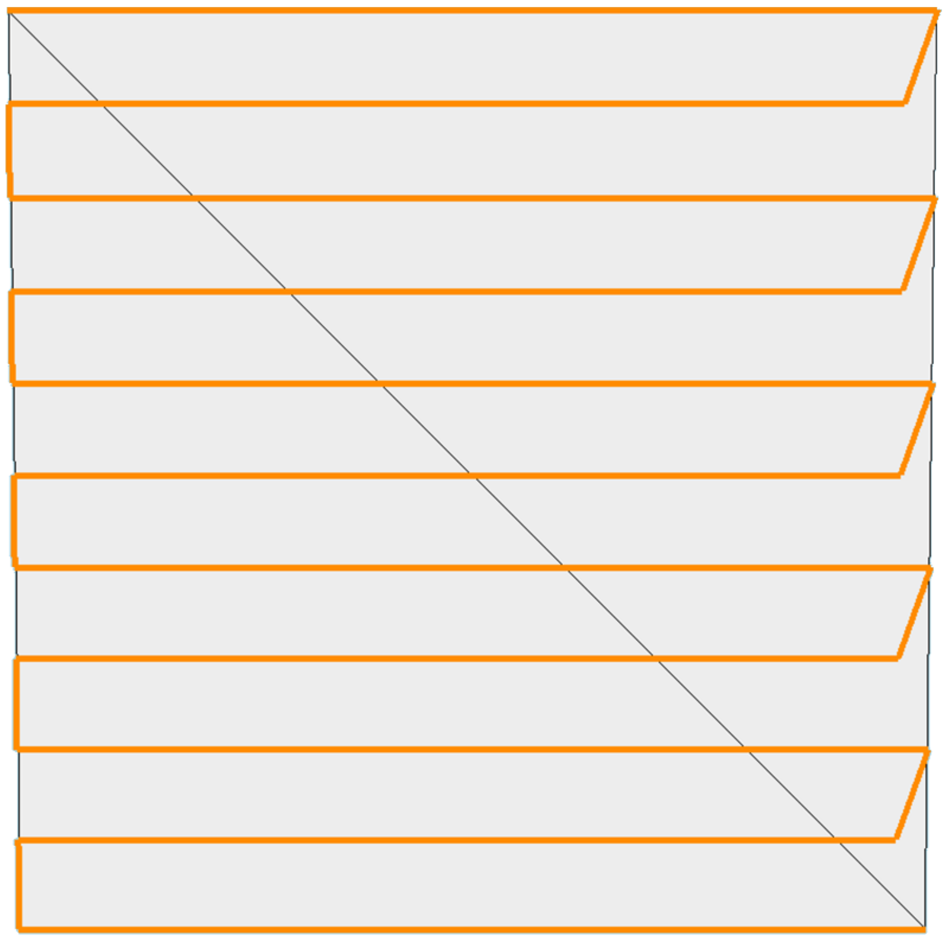


Figure 5.8: Flat reference panel within the custom path-planner

planning results. In a practical environment, the camera would be placed on a secondary segment of the end-effector, with the transform away from the end-effector center point known, where a preliminary scan could be conducted for obtaining a reconstructed mesh of the MUT. It was hoped for this report to conduct automated scans around samples to generate multiple point cloud images to be reconstructed, but due to various setup complications regarding obtaining the exact user/world coordinate position of the end-effector and also lack of user socket messaging during the time of obtaining results, this idea will remain outside the scope of the project. Instead, most images will be taken from one or two positions. Multiple acquisitions will use registration techniques such as center point transformation, ICP, and a combination of both.

Thre different types of scans were conducted. The first test was alongside boxes, where a low-

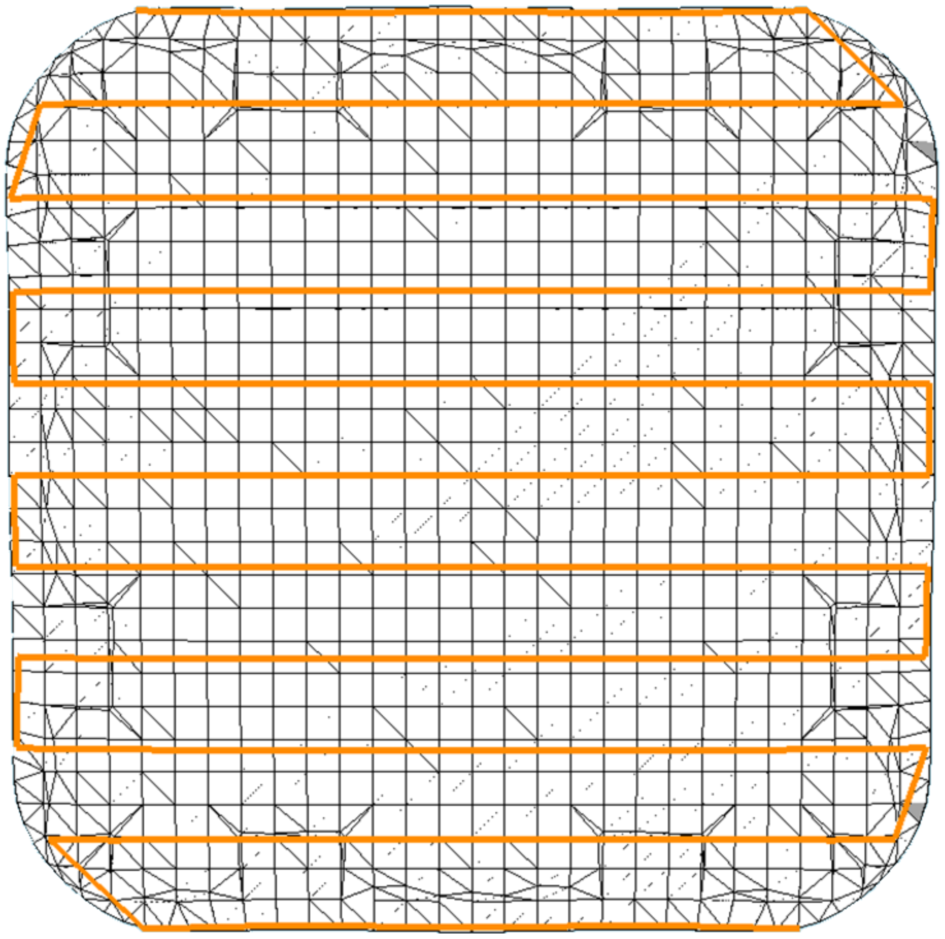


Figure 5.9: Reconstructed panel within the custom path-planner

risk solution for position testing is available without damaging any samples or the camera itself. The second and third tests used three different types car pieces, labeled as car1, car2, and car3. For the second test, the car pieces were placed on top of a flatbed where the camera was placed above the samples. For the third test, the car pieces were placed on a vice where the camera was placed facing the samples.

To obtain the robot end-effector coordinates, which in short were not reliable to obtain on the Fanuc iPendant due to a mastering issue preventing joint/pose movements after calibration, RoboDK was used to simulate the forwards and inverse kinematic movements for the physical robot using a custom configuration based on the DH parameters of the robot arm. Such RDK parameters have been validated to be 1-to-1 with the robot through joint length definitions with

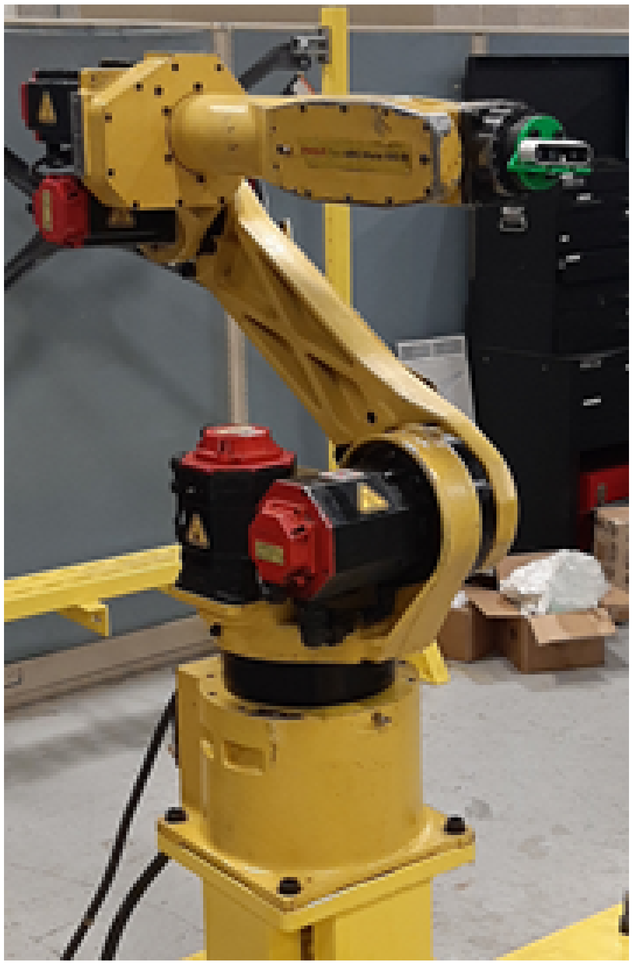


Figure 5.10: Camera as the end-effector of a Fanuc ARC Mate 100ib robot arm

Fanuc manuals, however even with this, shifting of point cloud images still occurs that could lead to 1 to a $2inch$ difference between meshes using center point transformation. It could be an intrinsic parameter of where the camera is, or some other factor. Validation for what the solution was time consuming and difficult without proper coordinates from the iPendant, hence why for these results, they are assumed to not be with respect to the robot's environment, as mentioned before. When proper validation methods can be conducted, then it will be possible to have a proper virtual-physical MUT corespondents between reconstruction software, the path-planner, and the inverse kinematics joint of the robot simulator to the physical robot. It should be noted that RoboDK figures showing the simulation is here avoided to prevent confusion since the custom robot setup did not have corresponding robot arm meshes, so only the path and possibly coordinates are shown.

Coordinates are tricky to show due to the high resolutions used shrinking each coordinate, so they were difficult to see and also obscured the path. Coordinates are only shown when greedy is used, since the path is more viewable by the gradient colors placed on each coordinate object, from the start at red and end at blue.

5.2.1 Box test

The first test is to attempt to virtually reconstruct a stack of three cardboard boxes within the robot environment and test of the end-effector can somewhat move towards the boxes, as seen in figure 5.1. Near-perfect accuracy between the virtual-physical correspondence was hard to validate, as mentioned why previously, but the purpose of this experiment is to show that there is still a close correspondence between virtual and physical placement of objects, hence why this test is still shown. This section will also show registration methods for center point transformation and Iterative Closest Point (ICP). Path planning is not examined for this test, though one waypoint was approximated for the touch test.

The camera was placed at two positions both directly facing the boxes, with a inverse kinematics solution for joints solved within RoboDK. The first position was placed at $T_{xyz} = [500, 0, 0]mm$ and $R_{xyz} = [0, 90°, 0]$ with respect to the base of the robot. The results of placing the camera at these two positions are seen in figure 5.11 where all three boxes are visible, and the second position placed $300mm$ upwards, at $T_{xyz} = [500, 0, 300]mm$ and $R_{xyz} = [0, 90°, 0]$ seen in figure 5.12 where the top two boxes, box1 and bo2, are seen. Note at position2, there is an RGB stretch on the bottom box which makes it only look like one piece of blue tape.

The first method was to use center point transformation by moving the second image $-300mm$ seen in figure 5.14. It can be seen that some shifting exists between these two models, but overall are closely aligned. Note that this is comparing meshes instead of point clouds, hence some parts overlapping. ICP was used in figure 5.16, where the Root Mean Square (RMS) value between the two point cloud boundary overlap of the two point clouds was 0.0192638 with the first mesh as the reference mesh and the transformation vectors applied to the second mesh,

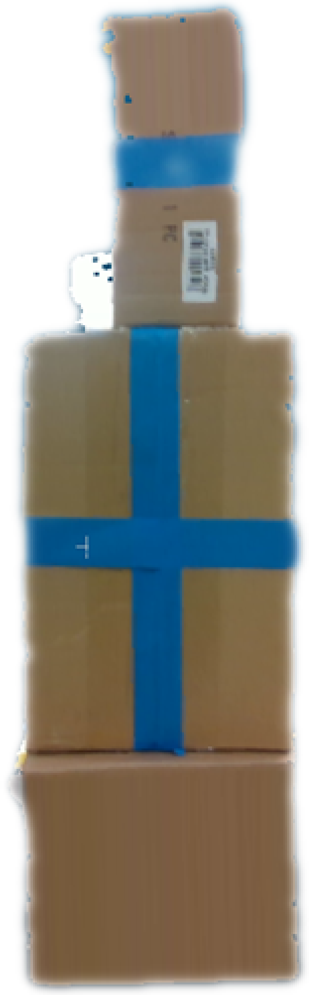


Figure 5.11: Box test camera position 1

$T_{xyz} = [0.035, 0.019, 0.018]mm$ and $R_{xyz} = [-2.0141°, 2.5899°, 2.7993°]$. It should be noted these transformation vectors are insignificant different compared to the $300mm$ change along z used in center point transformation. The results are shown in figures 5.15 and 5.16. It can be seen more clearly that the registered mesh (red) is not properly, since the bottom and middle boxes are "merged" together. A third test was conducted, where first the meshes would be aligned using center point transformation with the results seen in figure 5.14, then apply ICP with the first mesh used as the reference. The transformation vectors are $T_{xyz} = [92.572, 2.946, 30.528]mm$ and $R_{xyz} = [1.2542°, -0.2894°, -2.8632°]$, and the results are shown in figures 5.17 and 5.18. It can be seen that the two meshes are more aligned than in figure 5.16, though a metric to validate

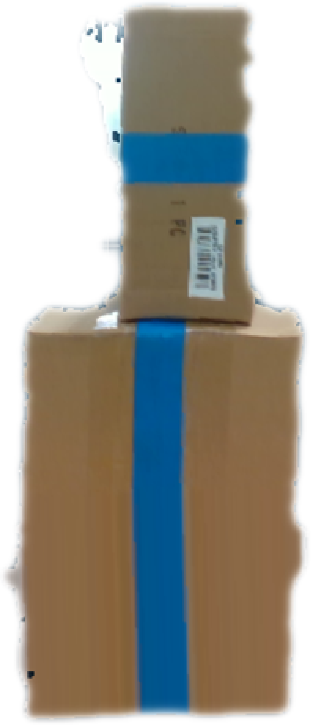


Figure 5.12: Box test camera position 2

accuracy is not definable for now. In any case, ICP can be used to further approximate meshes to align. A clear reference frame is needed to ensure proper virtual-physical correspondence however, and using ICP cannot be used to fix misalignment with virtual reference frame to it's physical correspondence. A test was conducted that would have the camera closely touch the top corner of the box by importing the final resulting mesh into RoboDK then solving the robot arm joints at the pose $T_{xyz} = [1300, 0, 250]mm$ and $R_{xyz} = [0, 90°, 0]$. Figure 5.19 shows the physical correspondence. As seen, the camera is approximately close to touching the boxes, but there is still a significant distance between. In the future, this will be worked out when it is easier to use the iPendent for obtaining user/world coordinate values.

5.2.2 Flatbed tests

As with the vice tests later, multiple car samples are used. Each car sample is made of different materials and have various shapes. The piece for car1 will also be used for a test where multiple camera

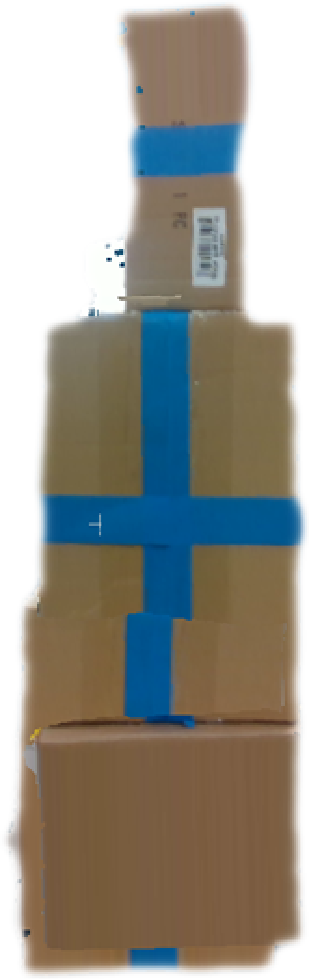


Figure 5.13: Box test camera positions 1 and 2 unregistered

movements are measured then transformed and reconstructed. Figure 5.20 shows the set up for the flatbed test, where the camera is placed above the MUTs at the pose $T_{xyz} = [1050, -250, 650]mm$ and $R_{xyz} = [-180°, 0, 180°]$ with respect to the base of the robot. Only a single mesh and point cloud image are obtained for this test. As seen later, the issue with using this method is that some of the meshes "merge" it's depth with the flatbed itself, so methods need to be used to extract the MUT from the table. The flat bed was examined without any samples placed on it, shown in figure 5.21. With more developed tools, a mesh like this could be used to remove the table itself from other point cloud images by removing any points within it's convex hull or similar shape. Since this was not immediately available, manual segmentation was used to extract the images instead. The table's position is static for each image.

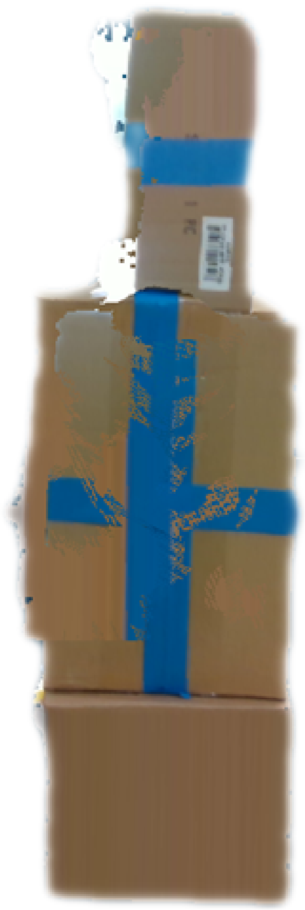


Figure 5.14: Box test camera positions 1 and 2 aligned using center point transformation

5.2.2.1 Flatbed car1

The first covered process will be for car1. Flatbed processes for car2 and car3 will follow the same procedure as explained here. First, a depth and RGB meshes is obtained, shown in figures 5.22 and 5.23. Using the RGB image as a reference, manual segmentation is used to cut separate the MUT and the flatbed, shown in figure 5.24. Finally, the mesh is exported into the custom path-planner which generates the nodes the end-effector with the NDE sensor should move towards. These nodes would then be exported into RoboDK to generate the joint positions for the physical robot.

For the scan, a subsection scan inside the piece is examined, where the ray plane which produces intersection lines onto the mesh has a resolution of 20×30 rays and a plane scale of $0.2 \times 0.15m$. Figure 5.25 shows the results of the zig-zag pattern, which the paths move alongside the mesh.

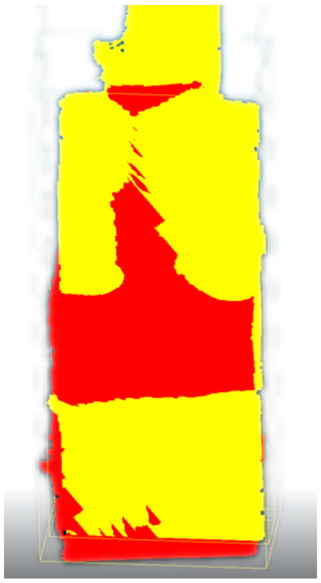


Figure 5.15: Box test camera positions 1 and 2 aligned using ICP, reference versus registered meshes

The greedy TSP is similar and therefore is not shown. The distance traversed was $3.081061m$. In comparison, the path along a plane would be $(20 * 0.15m) + 0.2m = 3.2m$. Between these two, these is a 3.71684% percent error. A better comparison would be the exact distance traversed on the physical sample itself, but a proper rubric for measuring out the distances along the physical curves would need to be conducted, which is outside the scope of these experiments. Since the position of the car pieces are placed in different positions with respect to the scanning environment, it is difficult to create a metric to compare this sample with the other car1 samples later. This issue also occurs for the later examined samples.

Also note that the top right corner should curve towards the camera, and the right side to the middle bottom has less thickness then the rest. This can be seen in the RGB image, but not in the segmented cloud. This is due to the depth difference not being significant enough for the camera. This issue persists with the other car samples for every method. It might be possible to fix this by obtaining multiple images of the sample, but this has yet to be tested. Future solutions for this problem are suggested later in the conclusion.

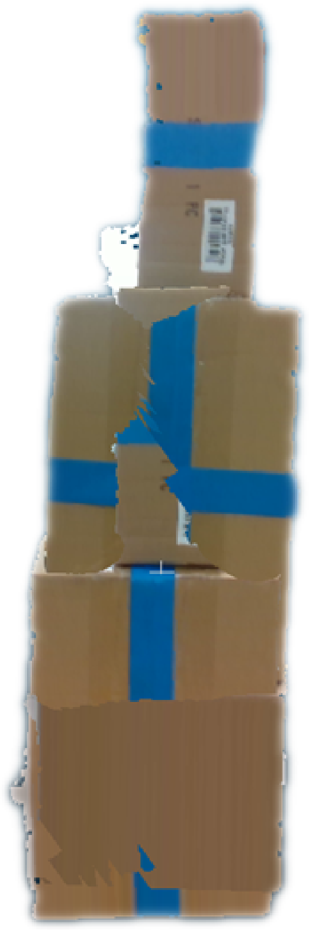


Figure 5.16: Box test camera positions 1 and 2 aligned using ICP

5.2.2.2 Flatbed car2

For car2, a similar procedure as car1 is conducted. Figures 5.26 and 5.27 show the depth and RGB meshes, and figure 5.28 shows the results after manual segmentation. As seen, there are holes inside of this model that are troublesome for path planning. This might be avoided obtaining and registering multiple point clouds to obtain the missing information. The path generated had a ray resolution of 20×50 and a scale of $0.5 \times 0.2m$, which should have generated a path along the entire sample. However, there was an error regarding large face amounts in which some ray intersections with the mesh would be detected. As a result, much of the path was cut off. The distance traversed was $2.1725001m$ in a zig-zag pattern, and the TSP algorithm provided similar results. As a similar metric to before, the ideal traversal distance on a flat plane would be $4.5m$, which provides a percent

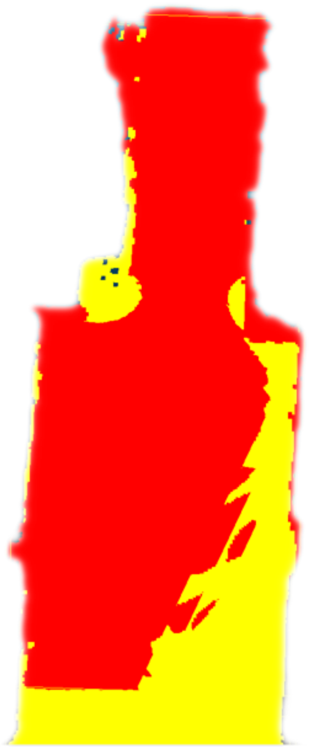


Figure 5.17: Box test camera positions 1 and 2 aligned using center point transformation then ICP, reference versus registered meshes

error of 51.7222%. The reason why high face density would cause issues was unknown and would most likely be a coding issue. There was also an issue with the code to run meshes with more than 32767 faces (the max value for a short). The solution for this was to lower the number of faces through decimation. This method was used when similar issues would arise for later samples. The mesh was decimated to 1000 faces, in which the same path was tested, shown in figure 5.30. The distance traveled was 3.744864m, which was the same for both the Zig-zag method and TSP. The percent error between the plane traversal and the mesh traversal is 16.7808%, which may be high due to larger depth traversal distances and some pieces not being traversed.

5.2.2.3 Flatbed car3

The third car piece follows a similar process as before. Figures 5.31 and 5.32 show the depth and RGB meshes, and figure 5.33 shows the results after manual segmentation. Again, there were holes

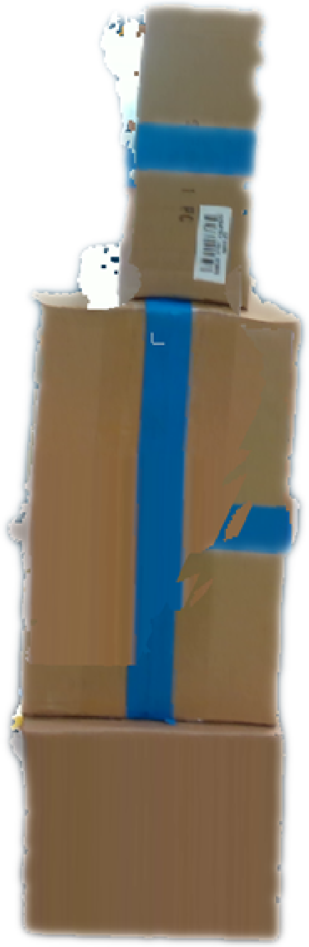


Figure 5.18: Box test camera positions 1 and 2 aligned using center point transformation then ICP

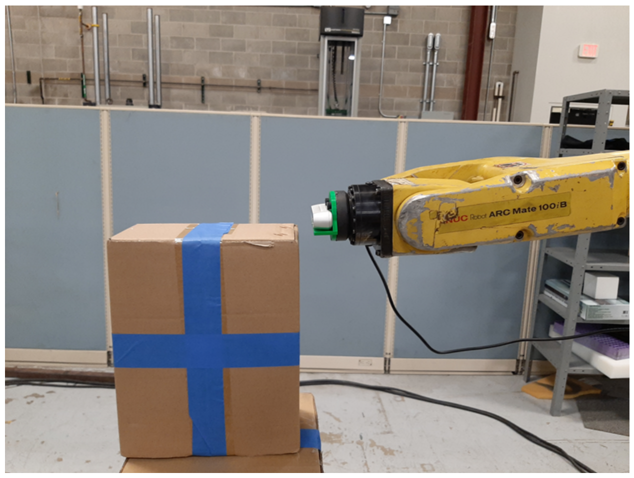


Figure 5.19: Box camera touch test

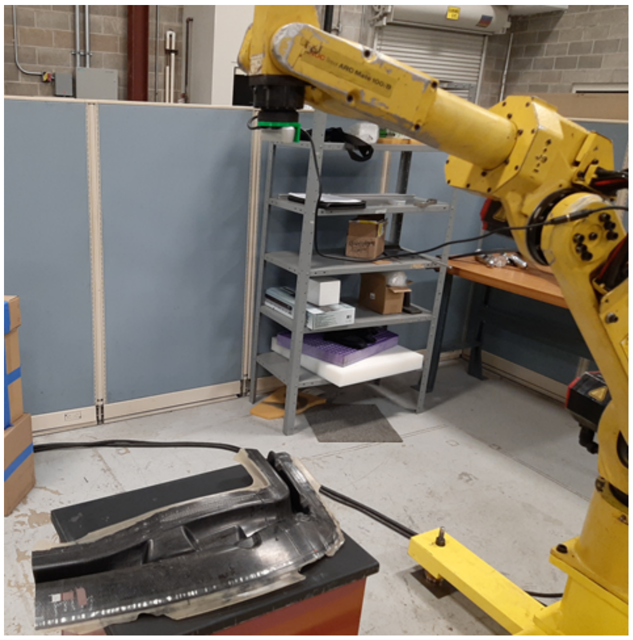


Figure 5.20: Physical setup for the flatbed test

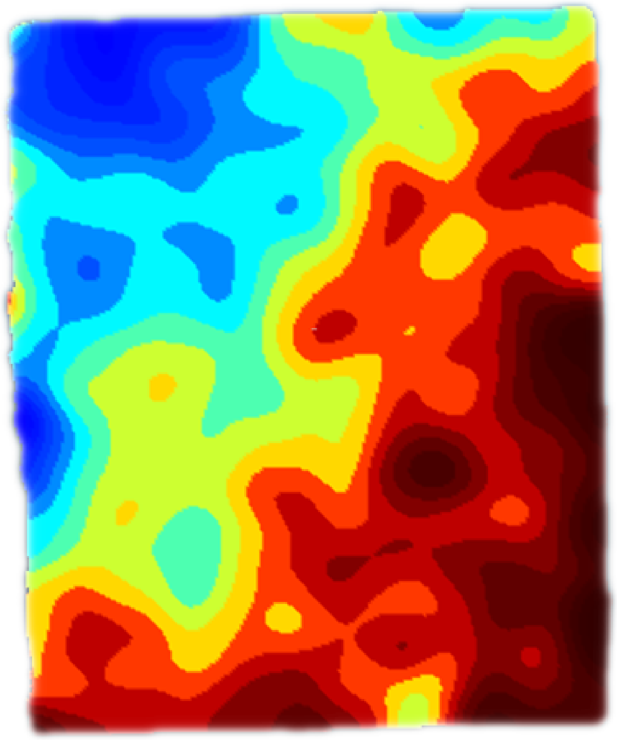


Figure 5.21: Flatbed no sample

in this model that would need to be filled. For placing into the path-planner, the model needed to be decimated as it was originally 75132 faces large, which as mentioned previously is too large where

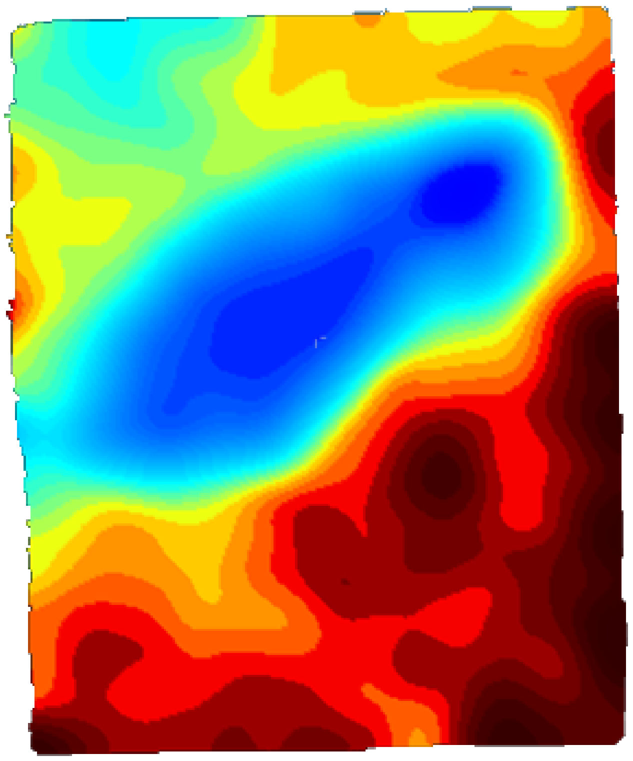


Figure 5.22: Flatbed car 1 depth mesh

less than or equal to 32767 faces are required. Figure 5.34 shows the path at a ray resolution of 30×40 and plane scale at $0.8 \times 0.5m$, covering the entire sample with intersections, with a traversal distance of $5.1664395m$. The greedy solution is identical to the zigzag solution. The same issue where too many faces prevents path generation. The sample was decimated to 1000 faces, seen in figure 5.35 using the same path parameters. The traversal distance using the same path parameter is $7.138192m$. Plane traversal approximation is not applicable here, since the path does not lie within a planar fashion.

5.2.3 Vice tests

The vice tests are similar to the flatbed tests where similar parameters will be used for each sample. The setups were shown earlier in figures 5.2, 5.3, and 5.4. Since each sample is moved in different position, there should be a direct relation between the two. In particular for car1 samples, the placement of the subsection scan will be different for each. However, comparing traversal distances

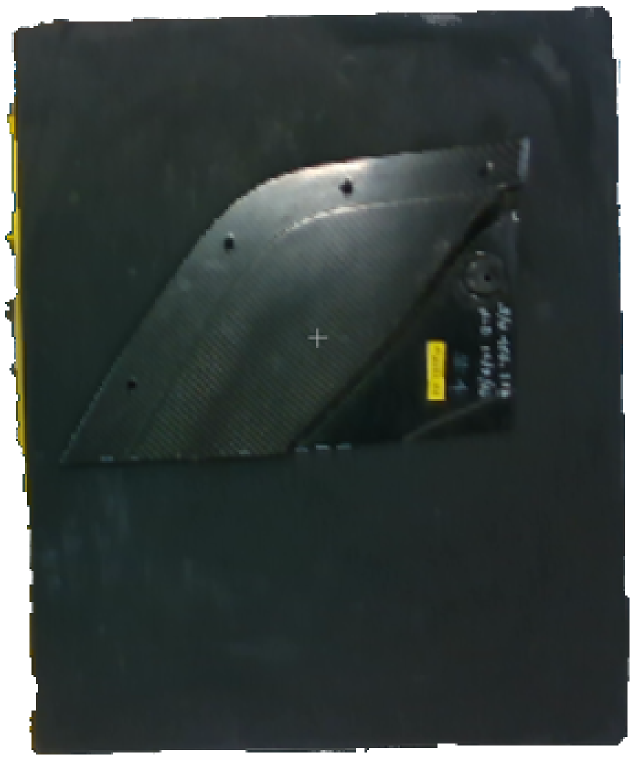


Figure 5.23: Flatbed car 1 RGB mesh

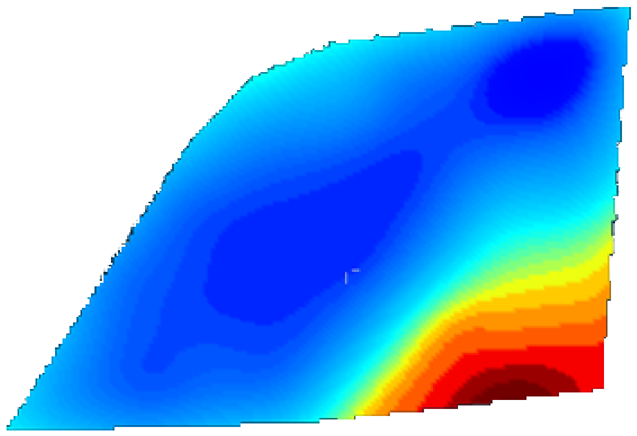


Figure 5.24: Flatbed car 1 depth cut mesh

and path generation performance might be useful for comparing overall performance. One other difference is that manual extraction is not necessary since there isn't an immediate background behind the MUTs. This means this method is slightly more simple and can be automated. The vices could be removed, but it does not majorly impact the results of the scan as long as if it is avoided in the path-planner. Figure 5.36 shows the depth mesh of the mesh, which similarly could

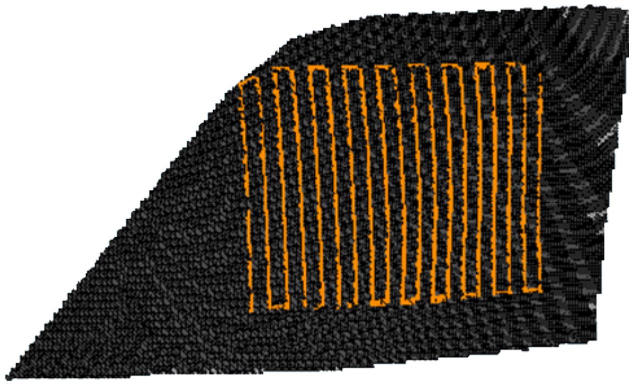


Figure 5.25: Flatbed car 1 path

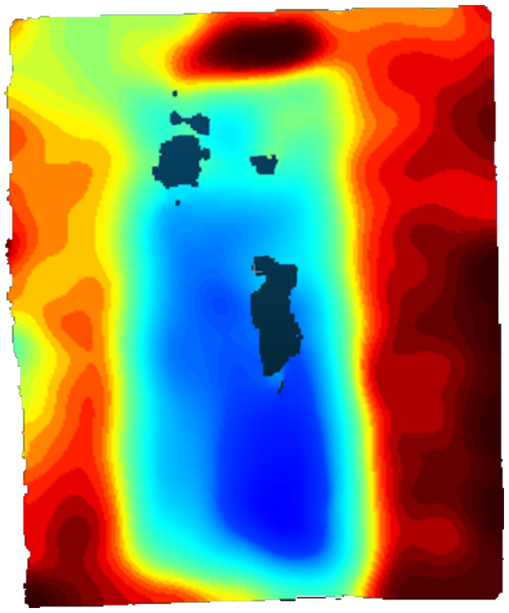


Figure 5.26: Flatbed car 2 depth mesh

be useful for removing the vice from reconstructed images. The vice's position is static for each image, similar to the flatbed.

5.2.3.1 Vice car1

The process for obtaining the mesh now only requires obtaining the mesh and possibly some post processing. Figures 5.37 and 5.38 show the depth and RGB meshes for car1 stabilized by the vice. Figure 5.39 shows the path at a ray resolution of 20×30 and a plane scale at $0.2 \times 0.15m$, similar to the flatbed test. The distance traveled was at $2.97711554m$, with the zigzag path similar to the TSP

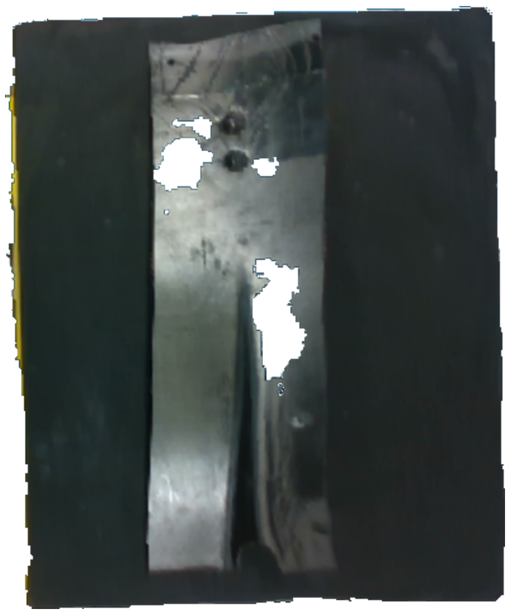


Figure 5.27: Flatbed car 2 RGB mesh

path. Using $3.2m$ traversal distance for the flat scan plan, there is a percent error of 6.96514%. This may be higher than previously due to the previous and later car1 meshes due to more curvature. It also seems to have a couples lines removed at the end of the scan.

5.2.3.2 Vice car2

The process is similar to before. Figures 5.40 and 5.41 show the depth and RGB meshes for car2. The depth distance can be clearly seen on the right side of these meshes, however a gap is formed due to point cloud shadowing. This leads to interesting results for the path planning. The ray resolution was set to 40×30 and the plane scale set to $0.7 \times 0.15m$, different from the previous flatbed test in order to avoid the vice. Note the model is rotated 90° clockwise from the previous image. This time, the zig-zag solution provides a different traversal path then the greedy solution. Both path types are seen in figures 5.42 and 5.43, where the start of the greedy path starts at the red and ends at blue, with a gradient between the two to determine traversal. The zig-zag path had a traversal distance of $3.6559842m$ while the greedy path had a distance of $3.5683413m$. Unfortunately for both these results, some of the path is cut off due to the high face density. Decimation was used to

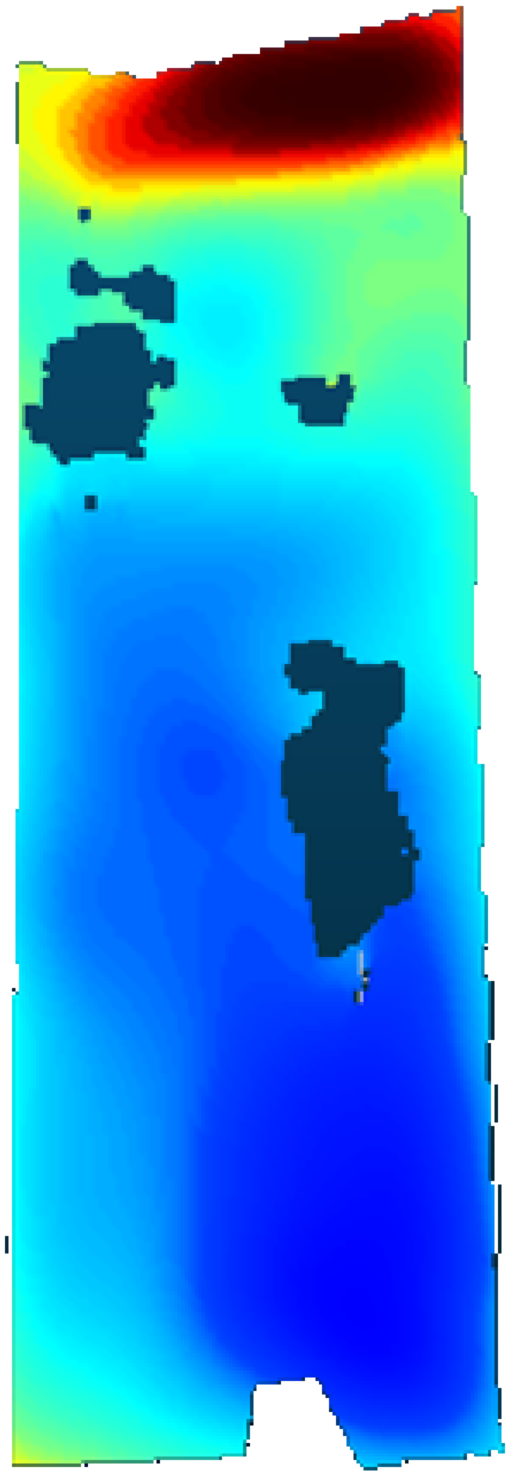


Figure 5.28: Flatbed car 2 depth cut mesh

create a mesh with 1000 faces, as seen in figure 5.44. The traversal path for the decimated mesh was 4.7542524*m*, with the zig-zag path the same as TSP path.

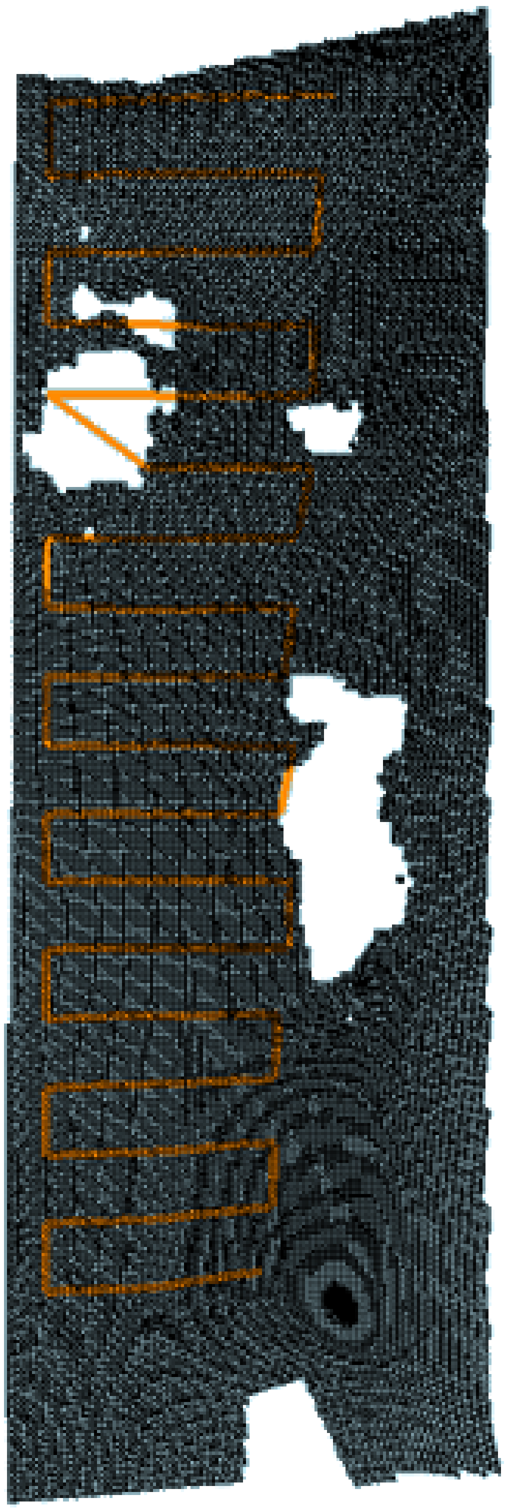


Figure 5.29: Flatbed car 2 path

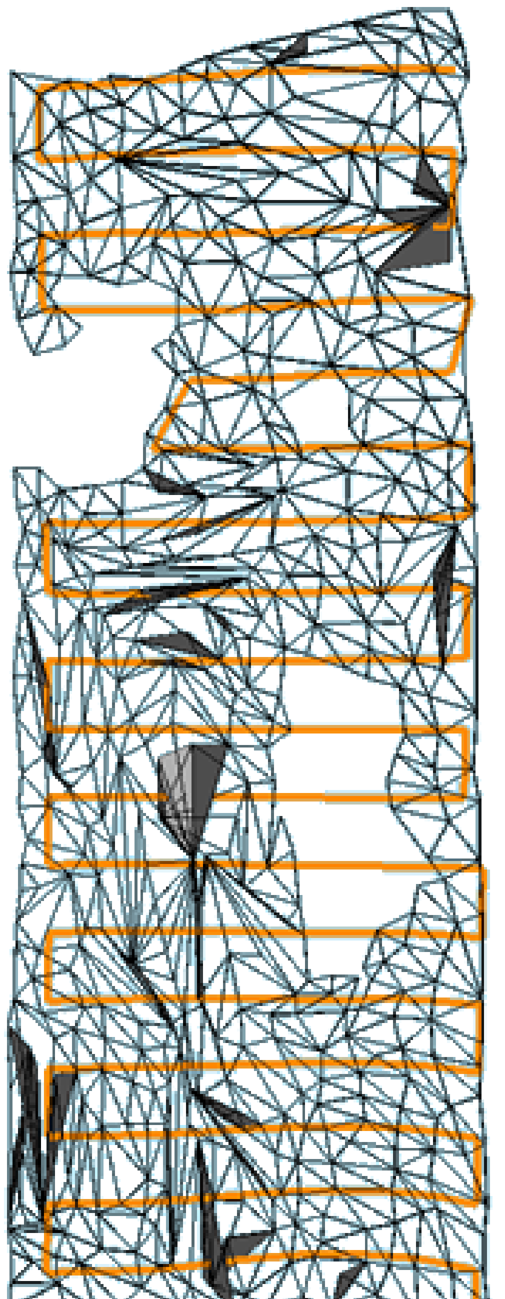


Figure 5.30: Flatbed car 2 path post decimation

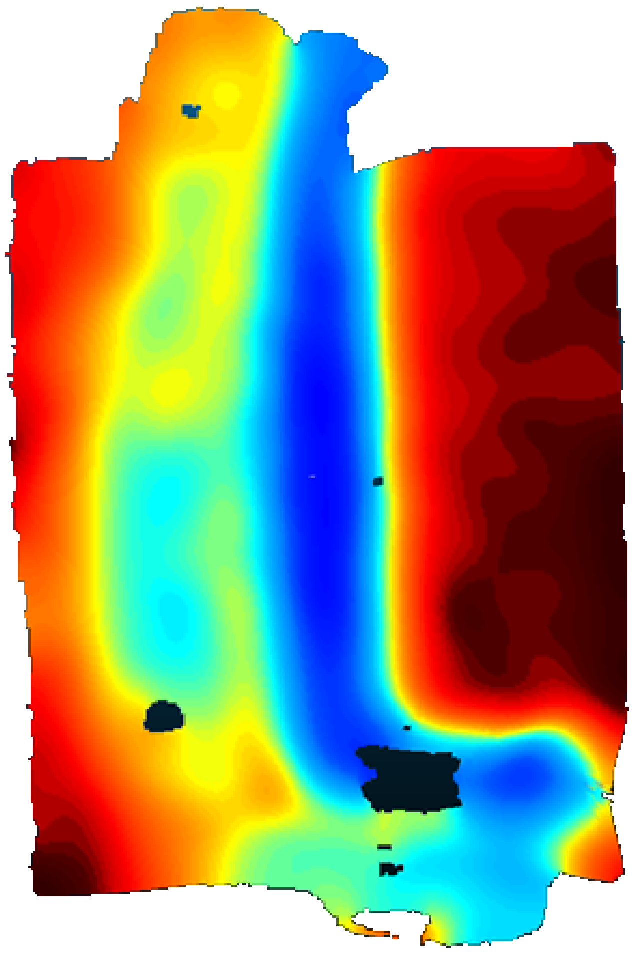


Figure 5.31: Flatbed car 3 depth mesh

5.2.3.3 Vice car3

Similar to the vice test before. Figures 5.45 and 5.46 show the depth and RGB meshes for car3. The opposite side of the mesh was scanned for this test. The mesh was also decimated to 1000 faces. A ray resolution of 30×40 and plane scale of $0.8 \times 0.5m$ was used, as seen in figure 5.47. The traversal distance was $6.8835125m$, with the zig-zag path the same as the TSP path.

5.3 Multi reconstruction: car1 measured translations

The purpose of this experiment is to show that multiple translations of the point cloud camera in a fixed environment using center point translation. First, the camera was swept from left to right alongside the car piece while facing directly at it for each position at $22in$ or $558.8mm$ away from

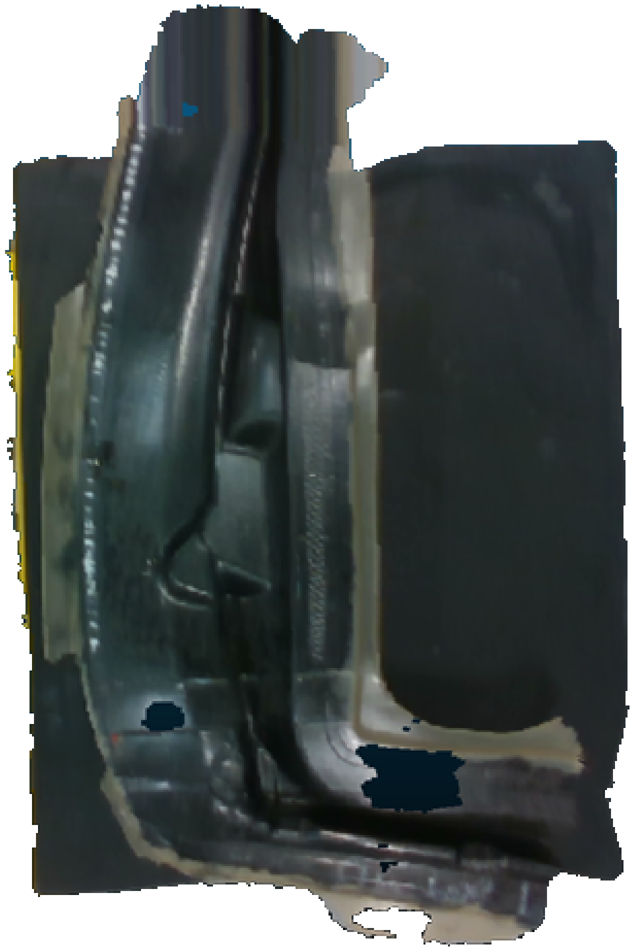


Figure 5.32: Flatbed car 3 RGB mesh

the sample. Fifteen images were acquired with 1 inch intervals, where the seventh image is used as the reference for all other meshes. The results are seen in figure 5.48.

The fifteen images were automatically transformed towards the leftmost image through the MeshLab script, then read into CloudCompare. The result is shown in figure 5.49. All background information including the room's walls, the table, and anything placed on the table were manually removed by vertex selection and removal methods for each image. The images were then merged together to form a single image, where the SOR filter removed any outlier points. Poisson reconstruction was applied with the octree depth at level 6 to prevent large numbers of bumping while maintaining good surface quality, as seen in figure 5.50. The output mesh was placed into MeshLab where a Laplacian smoothing filter was applied with 10 iterations. Note that this filter only smooths

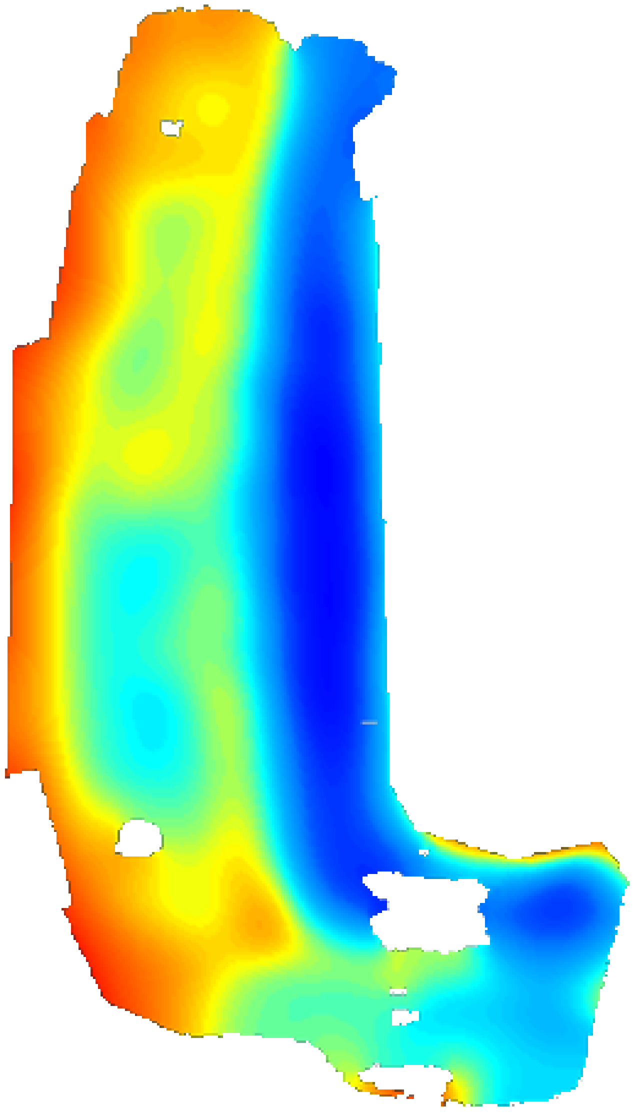


Figure 5.33: Flatbed car 3 depth cut mesh

geometric surfaces and not RGB components.

The model was placed inside of the custom path-planner. Only the zig-zag pattern was used as the greedy TSP yielded similar results. The resolution for the ray plane was set to 20×30 with a scale of 0.2×0.15 on the car piece, as used before. The resulting traversal distance was $3.2626913m$. If compared to a similar scan with a plane with $3.2m$. The rays were placed onto the sample as a subsection of the model. In comparison to this plane, there is a percentile error of 1.9591%. This is less than other, since this mesh was more flat than the previous car1 models.

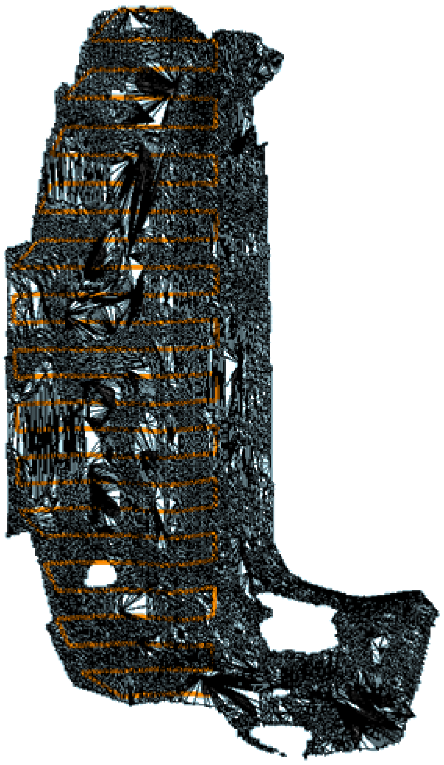


Figure 5.34: Flatbed car 3 path

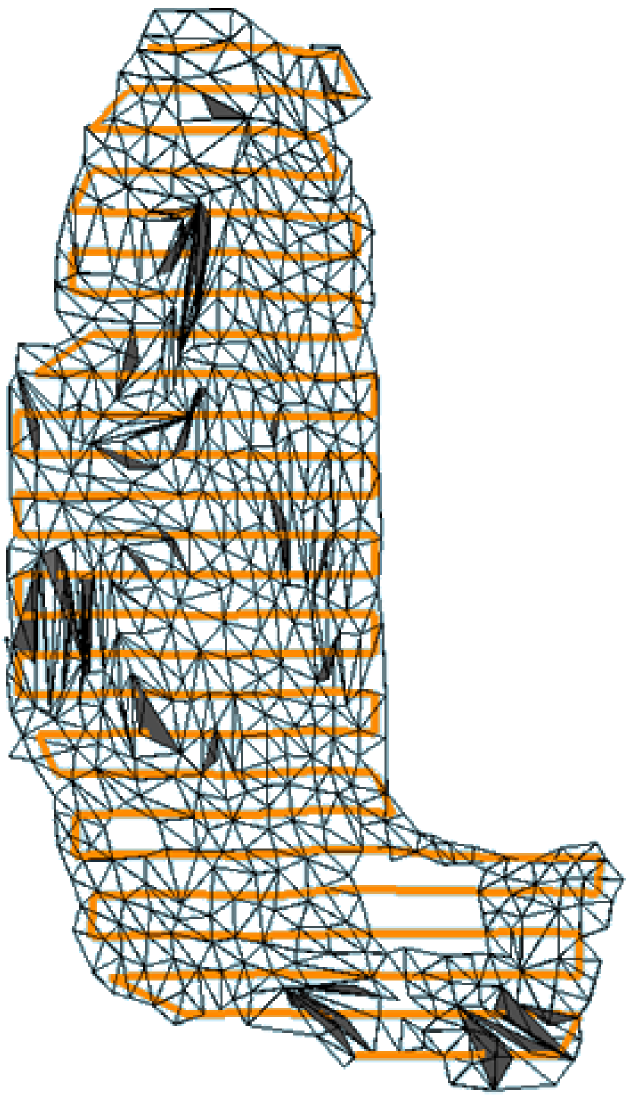


Figure 5.35: Flatbed car 3 path post decimation

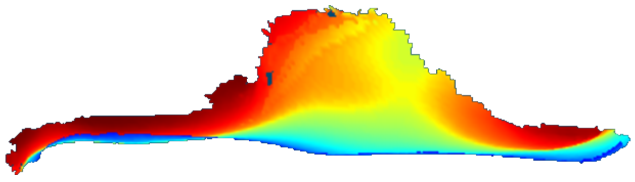


Figure 5.36: Vice no sample

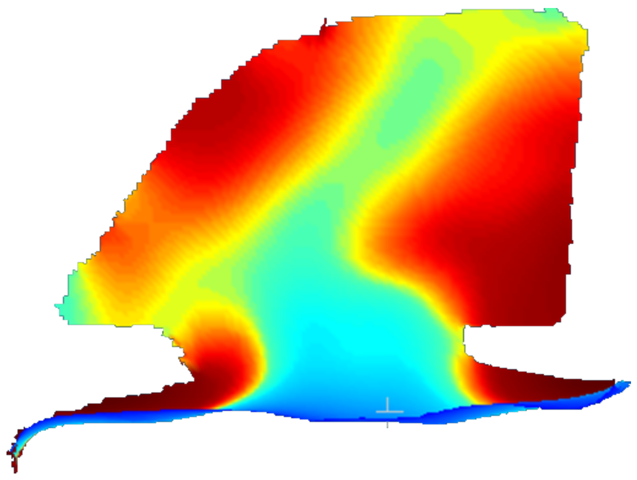


Figure 5.37: Vice car 1 depth mesh

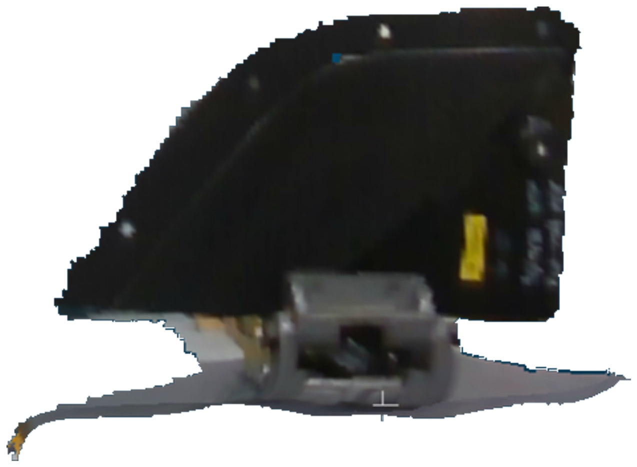


Figure 5.38: Vice car 1 RGB mesh

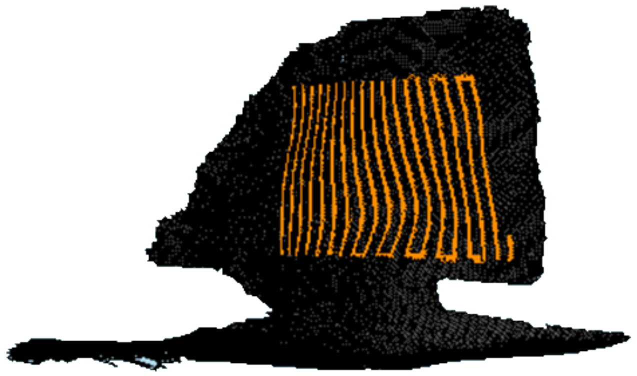


Figure 5.39: Vice car 1 path

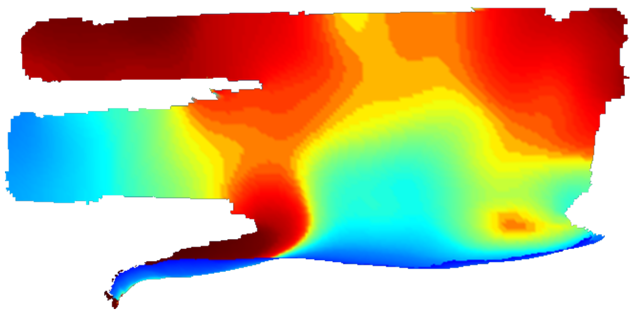


Figure 5.40: Vice car 2 depth mesh

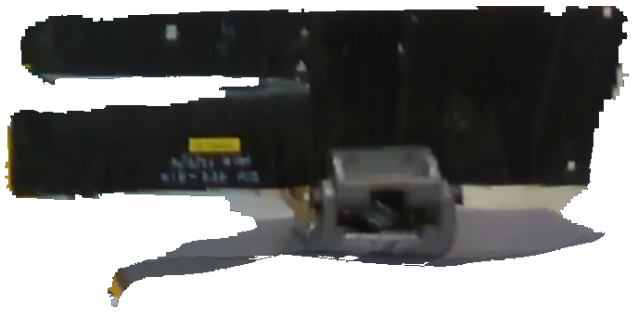


Figure 5.41: Vice car 2 RGB mesh

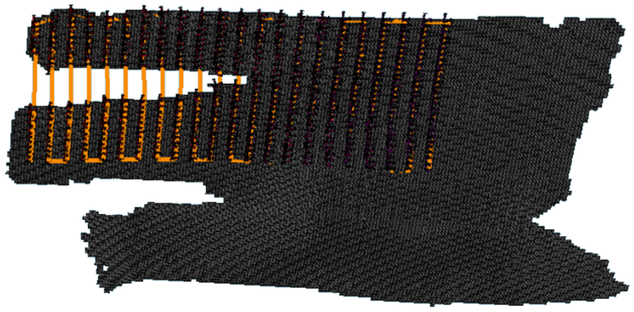


Figure 5.42: Vice car 2 path zig-zag

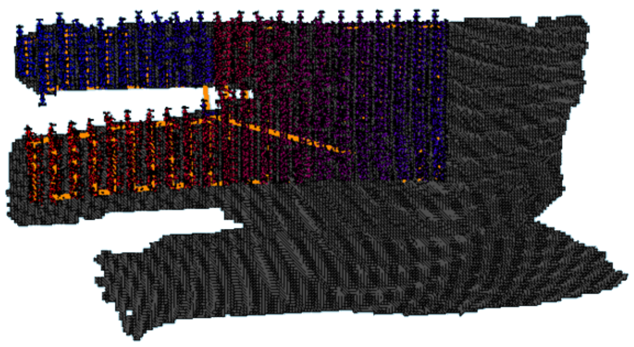


Figure 5.43: Vice car 2 path greedy TSP

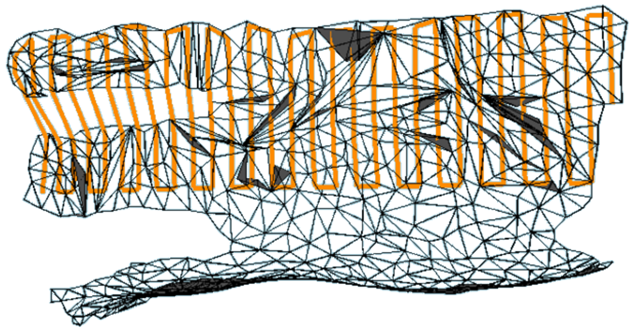


Figure 5.44: Vice car 2 path post decimation

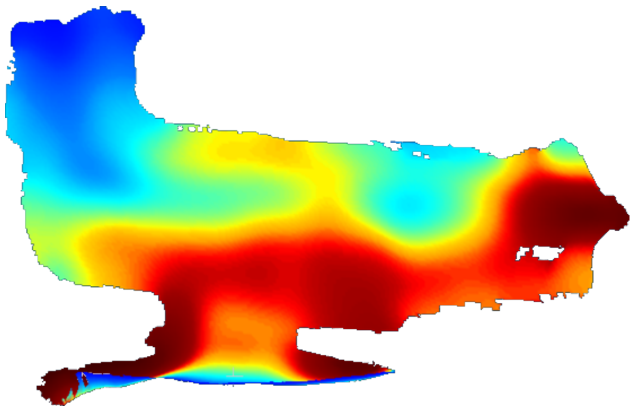


Figure 5.45: Vice car 3 depth mesh

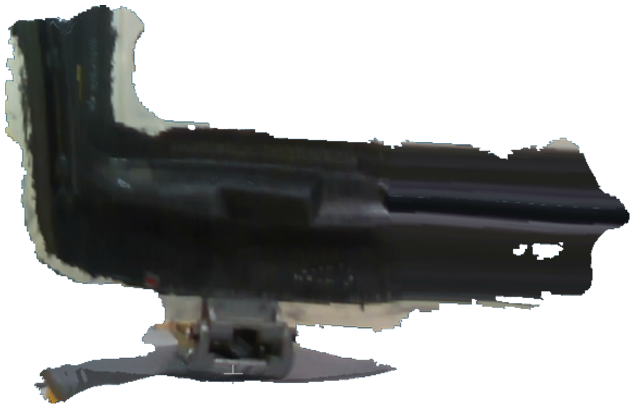


Figure 5.46: Vice car 3 RGB mesh

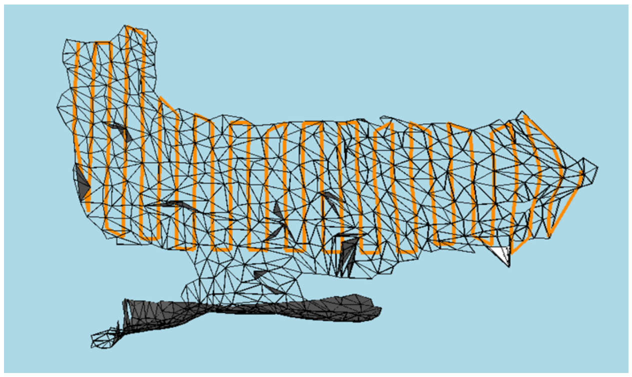


Figure 5.47: Vice car 3 path post decimation

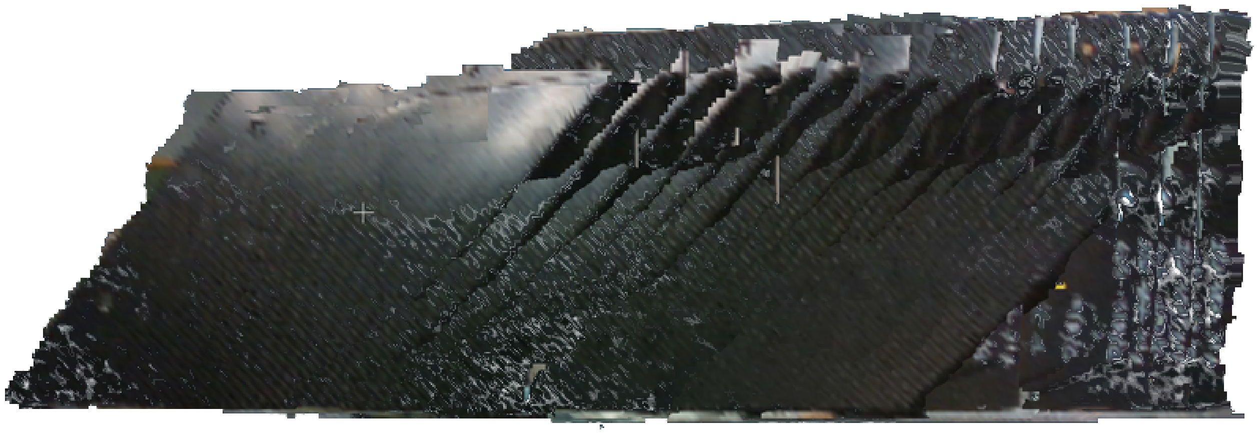


Figure 5.48: Unregistered car1 meshes

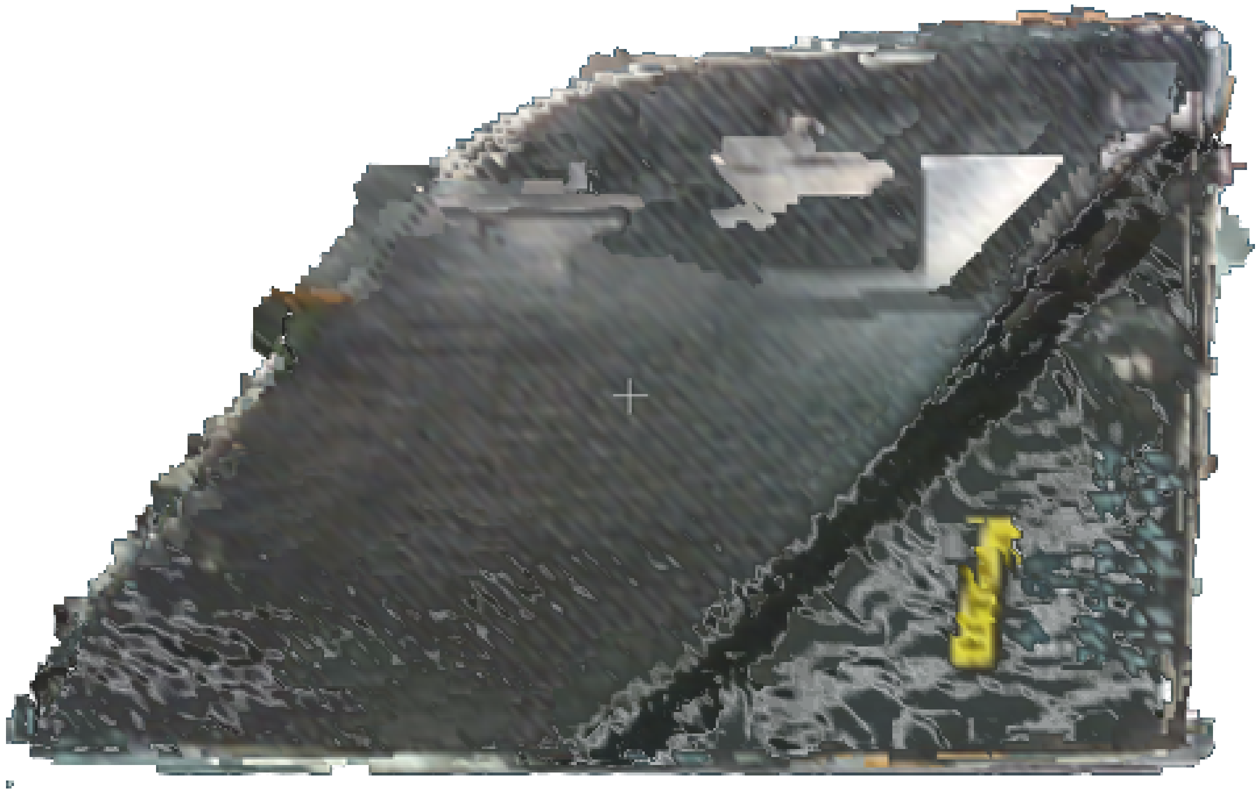


Figure 5.49: Aligned car1 meshes

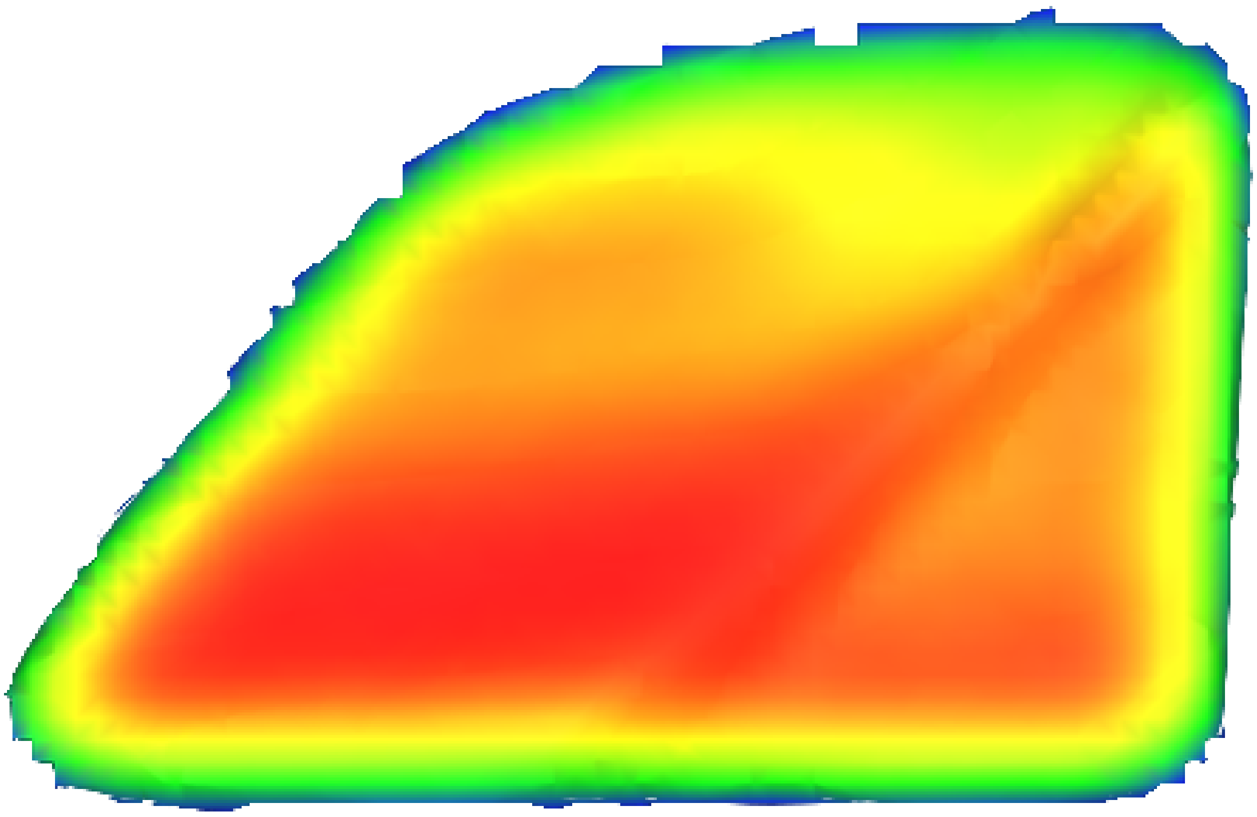


Figure 5.50: Poisson reconstruction on car1 vertices

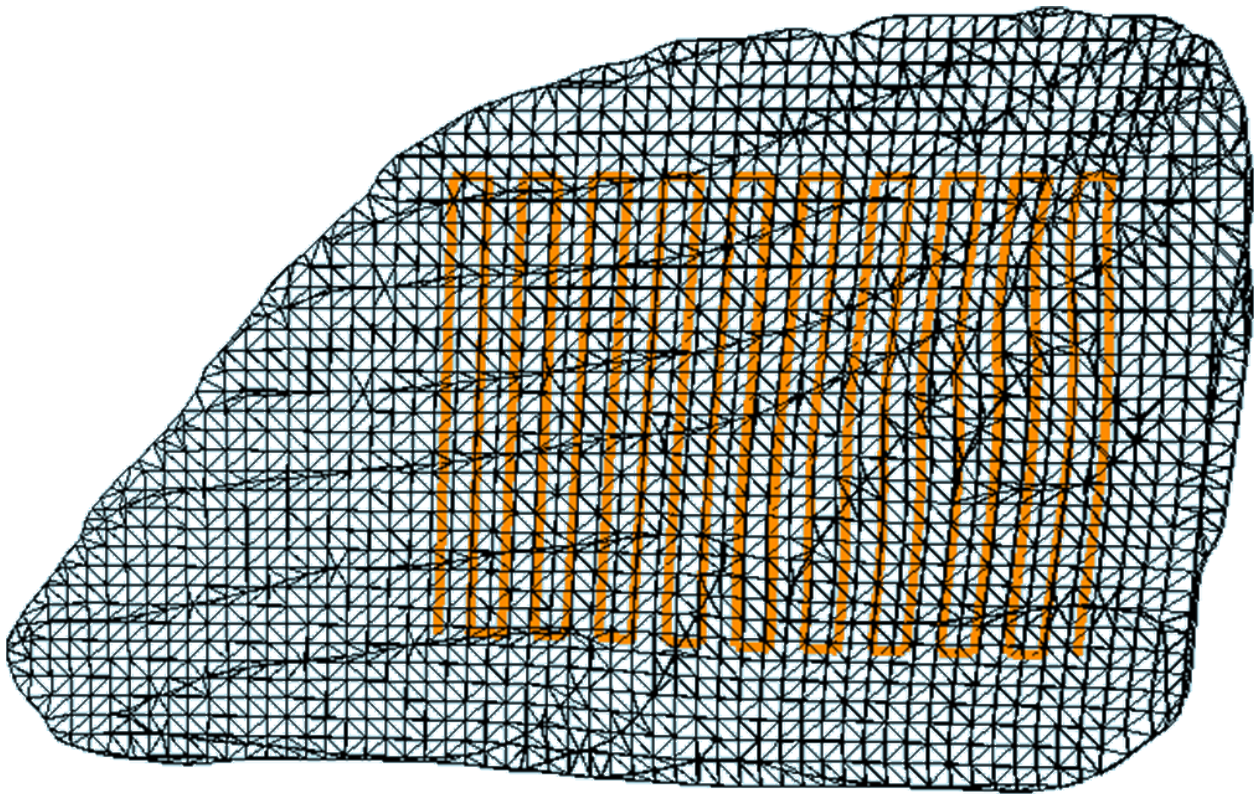


Figure 5.51: Path generation car1 15 models reconstruction

CHAPTER 6

CONCLUSION

Overall, the reconstruction and path planning processes are mostly complete for future robotic arm scanning usages. The virtual system explained in this report is an important stepping stone for automating complex NDE actuation systems within physical space. Further implementation of this system will focus on physical setups with variances in type of NDE techniques or probes used, different materials used, and variances geometrically complexity. Also more ideas for post processing, such as for defect detection within $3D$ space may be examined. This finalized system should open up new doorways for various types of researches or even implementations within practical environments. The future system might mean a couple reworks on the framework left from the base system, depending on what types of researches that are needed, but the purpose is to set a framework which can be morphed into different implementations later on. With better robot implementation or multiple stereo cameras for usage, $3D$ reconstruction around a sample is more possible.

There are still several improvements that should be implemented within both systems that have been mentioned throughout the report. First is to examine more methods for reconstruction, including using different stereo cameras or even different geometric parsing techniques. There are several morphing issues with the camera that makes depth validation difficult to assume. As such, virtual mesh to physical geometric correspondence testing should be developed and implemented. Sometimes image morphing or background components would create clusters that were difficult to resolve from automated means, and were manually moved. The purpose of the system is to provide as close to automated scanning as possible, so manual cleaning of meshes are not wanted. More general post processing techniques or improvements in the point cloud acquisition could help. It could also be possible to run a prior scan of the environment without the sample and remove points from an image with the sample based on the prior image. In the Poisson reconstruction process, where scalar field parameters are used, this is a manual selection process to approximate the best solution for the reconstructed mesh. The "best solution" should be handled automatically, in order to preserve qualitative results and remove manual integration.

For improvements on toolpath generation, more toolpath generation utilities and techniques can be develop to further improve robustness of the system. Only zig-zag and greedy traveling salesmen solutions were developed and covered, as well as a brief discussion of the hemispheric path planner for camera-as-end-effector scanning. More can be developed, such as a spiral point generation technique or a dynamic solution for exact traveling salesmen problems. Mesh edges need more consideration as well. If a ray-triangle intersection point lays on the edge, it's difficult to determine which normal between the two faces should be known, when the normal will be related to the two faces. As such, if a plane-edges intersector were to be developed, which would have a defined resolution with x planes intersecting the mesh, and points only y generated on edges, then implementing normal generation on edges would be required. With this new intersection configuration, it would be more robot to obtain points alongside meshes and avoid redundant points on shared intersected faces. The ray-triangle intersection approach is still viable to provide more control of resolution. For instance, if a mesh has a large amount of faces and a larger computation is required, then plane-edge intersection might be cumbersome. Decimation on input meshes was used to fix some path planning bugs, where too many faces would create conflicts for path planning. For meshes that require large facial counts, this might need to be fixed.

There might be improvements for determining a more proper weight between points rather than just using distance. Another rubric that seems more likely is to use time. Time is a more difficult metric, as it requires many properties from the robot actuator to be more valid. On top of this, it adds more computation time that might become significant depending on toolpath algorithms used. It is possible to import tools from RoboDK's *C#* module for this to become more possible. Otherwise for this report, this was a bit more to implement, and hard to validate without physical operation of the robots. Collision detection and prevention improvements could be implemented within the path planner. Basic approaches such as generating a ray between two waypoints and checking if collision between the ray and mesh occur could be used. Also integrating RoboDK, which can simulate robot movements, where the end-effector is, and provide collision detection, would be very useful. General visualization fixes could be examined that occur due to 3D engine issues, such as paths showing under meshes. When dealing with multiple segmented scans, or in the case of the custom pathplanner paths generated from a combination of different approaches, it might be best to use an $A*$ algorithm to determine the best paths for these scans. Multiple segmentation were also not implemented in the path planner code, but still could be used through multiple imports into RoboDK.

More usage of SprutCAM would also be advantageous. This is due to having very powerful tools for integrating robotic components around a determined environment and $5D$ path planning around these considerations. It was avoided for this report due to less customization and certain bugs with reconstructed models that was difficult to validate before computation if paths would be usable or not. Plus, having a custom pathplanner allows for more automated scanning with the current setup. In any case, SprutCAM is considered for future usage.

Physical robot complications were prevalent, mostly due to time restrictions caused by the Covid-19 pandemic that delayed the development of integrating the developed system with the Fanuc arms. The largest issue was that the robot could not be automatically moved due to safety concerns, nor controlled by RoboDK due to lack of user-socket messaging at the time. There is also a mastering and calibration issue that causes the robots to not move when calibrated, which was attempted to be fixed but was unable to be before the submission of this report. Becomes of these problems, validation of what the end-effector was in virtual space was difficult and time consuming due to the process of obtaining joint positions from simulation data from RoboDK to physically reach a point with the Fanuc robot. These issues are to be shortly resolved, in which proper validation of robot end-effector position can be known. From this, automated reconstruction techniques using the camera-as-end-effector can be used, and physical NDE scanning can be done.

Proper error measurement tools for validating proper formation of meshes should be developed or integrated. The issue with reconstructed models, especially ones that required multiple images that were aligned, would contain difficulty to denoise the image and afterwards keeping integrity of the formation of the mesh. The idea for this is to conduct a second automated scan that uses an accurate depth sensor, such as an ultrasonic transducer which could also be used alongside the NDE scan, to obtain depth data around the MUT. The first scan using stereo cameras would be used as a reference for this scan. There then can be a comparison between the two models, or even the second result from the depth sensors being used for path planning. Another possibility is to use predefined models to compare with reconstructed models. These were not readily available for most tests and hence were not used.

The virtual system is still a basis in which is required to be designed and deployable before the physical system can operate. In it's current setting, it should be possible to obtain mostly automated results of a dynamic range of samples, leading into robust possibilities of obtaining NDE scans from various mechanical and civil fields. When the CPS is operational, this could also lead into more research into other robotic NDE integrated fields, such as mobile robots in terms of visualization and post processing information. Despite more improvements needed, the system alone should have an impact in terms of NDE research and deployability.

Future considerations also need to consider interfacing the actuation and NDE systems, conducting physical scans, and post processing. Not much for improvements are explained here since results are not shown. The most important obstacles for these are to ensure virtual-physical interface validation so sensors or samples do not become damaged. This requires improvements on the Fanuc arms and several end-effector pose tests within a calibrated setting. Later, a dual robot system will be used, in which a pitch-catch configuration between two transducers will be used. This configuration will require heavy synchronization between the two robots to avoid losing gain or even the entire signal due to bad alignments of the sensors. For post processing, NDE reconstruction is considered, where point cloud data from NDE scans are compiled into a mesh, could be considered. This would have the same processes as MUT reconstruction. Finally, more focus on real-world applications rather than within a controlled laboratory setup should be considered. Controlled environments do not provide perfect conditions for the sensor, considering that other environmental factors such as temperature may effect the results of the sensor. In this systems current state, there is some development that is required for an effective system. The results of this paper hopefully convey that most components before physical scanning are aligned, in which future developments would further improve results.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Dave Tong Anders Grunnet-Jepsen. Depth post-processing for intel realsense depth camera d400 seriess.

[2] Intel Corporation. Post-processing filters.

[3] Magdalena Czaban. Aircraft corrosion – review of corrosion processes and its effects in selected cases. *Fatigue of Aircraft Structures*, 2018, 05 2019.

[4] Lyle McGeoch David Johnson. The traveling salesman problem: A case study in local optimization. *Local Search in Combinatorial Optimisation*, page 215–310, 1995.

[5] Joey de Vries. *Learn OpenGL*. 2015.

[6] Ugyen Dorji and Reza Ghomashch. Hydro turbine failure mechanisms: An overview. *Engineering Failure Analysis*, 44:136–147, 2014.

[7] Eduardo S. L. Gastal and Manuel M. Oliveira. Domain transform for edge-aware image and video processing. *ACM TOG*, 30(4):69:1–69:12, 2011. Proceedings of SIGGRAPH 2011.

[8] Charles Hellier. *Handbook of nondestructive evaluation*. McGraw-Hill, 2001.

[9] David Jeske. Simplescene: 3d scene manager in c-sharp and opentk, 2014.

[10] Michael Kazhdan and Hugues Hoppe. Screened poisson surface reconstructionn. *ACM Transactions on Graphics*, 2013.

[11] Matthew Bolitho Michael Kazhdan and Hugues Hoppe. Poisson surface reconstruction. *Eurographics Symposium on Geometry Processing*, 2006.

[12] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[13] Yaron Lipman Marc Alexa Christian Rössl Olga Sorkine-Hornung, Daniel Cohen-Or and Hans Peter Seidel. Laplacian surface editing. *Eurographics Symposium on Geometry Processing*, 2004.

[14] Sehba Siddiqui Annies Minu Vivian Lobo, Blety Babu Alengadan and Nazneen Ansari. Traveling salesman problem for a bidirectional graph using dynamic programming. *2016 International Conference on Micro-Electronics and Telecommunication Engineering (ICMETE)*, pages 127–132, 2016.