ERDC/GRL TR-23-5

US Army Corps
of Engineers®
Engineer Research and
Development Center

Dambot™

Leveraging MOVEit for Object Inspection in Simulation

Nathan Dodson, Mike Paquette, and Garry Glaspell

November 2023

Geospatial Research Laboratory

The US Army Engineer Research and Development Center (ERDC) solves the nation's toughest engineering and environmental challenges. ERDC develops innovative solutions in civil and military engineering, geospatial sciences, water resources, and environmental sciences for the Army, the Department of Defense, civilian agencies, and our nation's public good. Find out more at www.erdc.usace.army.mil.

To search for other technical reports published by ERDC, visit the ERDC online library at https://erdclibrary.on.worldcat.org/discovery.

ERDC/GRL TR-23-5
November 2023

Leveraging MOVEit for Object Inspection in Simulation

Nathan Dodson, Mike Paquette, and Garry Glaspell

US Army Engineer Research and Development Center (ERDC)
Geospatial Research Laboratory (GRL)
7701 Telegraph Road
Alexandria, VA 22315-3864

Final Technical Report (TR)

Prepared for Headquarters, US Army Corps of Engineers
Washington, DC 20314-1000

Under PE Number 29785D, Project Number 497149, Task Number 8, “Dambot™”

Abstract

Herein we evaluate using a robotic arm with an attached camera to investigate objects of interest in simulation. Specifically, a Husky unmanned ground vehicle with a Panda Powertool was used in the simulation. The code enabled an operator to initiate a preconfigured set of motions when an object of interest was identified. The scan was stored in a database file that was used to generate a 3D mesh of the scanned object. The report describes both setting up the simulation and the code used to scan objects of interest.

Contents

Abstract ii

Figures and Table iv

Preface v

1 Introduction 1

1.1 Background 1
1.2 Objectives 1
1.3 Approach and Scope 2

2 Payload Setup 3

3 Manipulator Movement Planning 5

3.1 Semantic Robot Description Format File (SRDF) Creation 5
3.2 Path Planning Functions 5

4 Inspection Toggling 7

4.1 Evtest 7
4.2 Toggle Controls 8

5 Real-Time Appearance-Based Mapping (RTAB-Map) Data Recorder 9

5.1 Simulated Object Inspection 9
5.2 Object Inspection 9
5.3 Postprocessing 12

6 Conclusion 15

Bibliography 16

Appendix A : Movement Functions 17

Appendix B : Toggling Code 22

Abbreviations 24

Report Documentation Page (SF 298) 25

Figures and Table

Figures

1. Payload setup on Husky unmanned ground vehicle (UGV). 4
2. Floorplan of TurtleBot3 House. 10
3. Bookshelf in TurtleBot3 House. 11
4. Trash can in TurtleBot3 House. 11
5. Occupancy grid of a bookshelf in 2D. 13
6. Mesh of the bookshelf in 3D. 13
7. Occupancy grid of a trash can in 2D. 14
8. Mesh of the trash can in 3D. 14

Table

1. Preconfigured movements remapped to joystick buttons. 6

Preface

This study was conducted for the US Army Corps of Engineers under Program Element 29785D, Project Number 497149, Task Number 8, "Dambot™." The technical monitor was Dr. Anton Netchaev, US Army Engineer Research and Development Center (ERDC), Information Technology Laboratory.

The work was performed by the Data Representation Branch of the Topography Imagery and Geospatial Research Division, US Army ERDC Geospatial Research Laboratory. At the time of publication, Mr. Vineet Gupta was branch chief; Mr. Michael Mailloux was acting division chief; and Dr. Austin V. Davis was technical director. The deputy director was Ms. Valerie L. Carney, and the director was Mr. David R. Hibner.

The authors would like to acknowledge the following individuals for their contributions to this project: Mr. Chuck Ellison, Mr. Steven Bunkley, and Mr. Jordan Klein.

The commander of ERDC was COL Christian Patterson, and the director was Dr. David W. Pittman.

This page intentionally left blank.

1 Introduction

1.1 Background

The recent influx of low-cost sensing technologies has caused a revolution in the field of robotics, enabling higher quality and new forms of data, resulting in more complex robotics functions. The field of remote sensing has greatly benefited from these sensors, which allow for denser information at a lower cost, in addition to a wide array of previously unavailable modalities. While this report focuses on visual inspection, nondestructive testing (NDT) is a natural extension, specifically in assessing structural integrity. Low size, weight, power, and cost (SWaP-C) sensors are now capable of collecting enough information about the surrounding environment to allow autonomous movement of an unmanned ground vehicle (UGV). This allows the warfighter, equipped with a remote-sensing robot payload, to perform reconnaissance roles in dangerous environments. For example, a UGV mounted with sensors can enter a building, construct a map of the interior, and transmit the data without placing the warfighter in danger. However, the ability to scan suspicious objects that may be present in a building is limited on a standard robot. Inspecting objects of interest requires higher quality, denser scans compared to the scans employed in mapping.

1.2 Objectives

The goal of this paper is to develop a solution that is capable of performing detailed autonomous inspections on objects while an operator teleoperates a robot in a simulated environment. This goal must be achieved on two fronts: software and hardware. The software package must allow the teleoperator to initiate a separate data collection, when necessary, on an object of interest. In addition, a hardware package that can maneuver a sensor with great precision around the object of interest must be developed. This research supports the "Revolutionize and Accelerate Decision Making" research and development priority, which states,

> USACE leverages data science, artificial intelligence, machine learning, robotics, trade-space analytics, geospatial engineering, and advanced models and simulations to revolutionize geospatially informed

decision-making by understanding the operational environment, predicting likely outcomes, decreasing uncertainties by 80 percent, and shaping faster decisions that minimize risk, reduce costs, and accelerate mission delivery by 5× for military, civil works, and disaster response operations.

1.3 Approach and Scope

A typical UGV, with differential drive, is restricted to movement in the xy-plane and rotation around the z-axis (i.e., yaw). Due to this limitation in movement, parts of a particular object of interest may be obscured in the sensor's view, especially if the sensor is rigidly mounted to the UGV. Therefore, a dynamic sensor configuration is preferred since it allows a greater range of motion. A robotic manipulator (i.e., arm), connecting the sensor to the UGV, can achieve this function. Robotic manipulators are capable of moving with six degrees of freedom (DoF), providing the sensor with a greater number of viewpoints from which to collect data. An ideal sensor for this application is a red-green-blue and depth (RGB-D) camera. RGB-D cameras are ideal for object inspection because they are low-cost devices that provide color and depth information. Operators can use this information to quickly assess the size and functionality of inspected objects.

Regarding the software, the package must enable the teleoperator to control and track placement (i.e., the viewpoint) of the RGB-D camera using the robotic manipulator. In a single instance, the operator should be able to launch all related packages for teleoperation and mapping. Several preconfigured manipulator movements are required and need to be activated using a controller. The operator can use the controller to toggle data collection on or off. This technique allows us to reduce file size without interfering with the primary mapping mission of the UGV.

In this effort, the Robot Operating System (ROS) (ROS 2021), a framework for writing robot software, was used. Gazebo (ROS wiki 2021), a robot simulation environment, was used to simulate the hardware. MOVEit, another ROS-based package, was used to manipulate the arm. Finally, the 3D point clouds were generated using Real-Time Appearance-Based Mapping (RTAB-Map) (ROS wiki 2022).

2 Payload Setup

As previously discussed, the desired payload is an RGB-D camera connected to a robotic manipulator that is mounted to a UGV platform. Since the scope of the report is solely simulation, one criterion for component selection is that the platform must have ROS support for Gazebo simulations. A suitable robotic platform that meets this criterion is the ClearPath Robotics Husky UGV (Clearpath Robotics 2018). The Husky platform is a medium-sized UGV that can easily be fitted with a robotic manipulator. ClearPath Robotics has created extensive repositories for simulating and controlling a Husky UGV within Gazebo using ROS.

There are considerably fewer ROS-enabled robotic manipulators that include Gazebo simulation functionalities. Fortunately, the Franka Emika Panda Powertool has several Gazebo packages developed by others in the ROS community, so this manipulator is the optimal choice for simulation (Franka Emika 2022). In addition, the Powertool has existing packages in MOVEit, a motion planning framework. The Panda Powertool has seven DoF and an 855 mm reach; this allows many viewing angles for an RGB-D camera attached to its end effector.

The Panda Powertool is relatively light, weighing only 3 kg. The payload capacity of the Husky is 75 kg. The Panda Powertool was attached to the center of the Husky's top mounting plate to give maximum reach in all directions. Furthermore, the Powertool and Husky must be facing the same direction, as pictured in Figure 1. The RGB-D camera was attached to the end of the manipulator after the gripper was removed.

The Unified Robot Description Format (URDF), a format based on XML, is used to represent robot models in ROS (ROS 2020). URDF files contain specifications of the robot's dimensions, sensors, and drive train. The individual parts of a robot, as well as the manipulator, comprise "link" elements and are connected to each other via URDF "joint" elements. Furthermore, when operating a robot in the Gazebo simulation environment, the URDF file must also specify density of the material used for inertial calculations and motor controller characteristics. Fortunately, all of the hardware components chosen have existing URDF files that are publicly available.

Figure 1. Payload setup on Husky unmanned ground vehicle (UGV).

3 Manipulator Movement Planning

The Panda Powertool must be controlled to move the RGB-D camera into useful positions. Creating a package solely for the purpose of controlling the joints of the Powertool is beyond the scope of this paper, and therefore an existing package is used to control the joint positions. MOVEit, discussed in the previous section, is able to create motion plans based on URDF files (Coleman et al. 2014). MOVEit also includes support for the Powertool in its tutorials, so existing code can be adapted to work with the entire hardware setup in Gazebo. These reasons make MOVEit the optimal choice for the motion planner for this application (Görner et al. 2019).

3.1 Semantic Robot Description Format File (SRDF) Creation

In addition to the URDF file, MoveIt requires a Semantic Robot Description Format (SRDF) file (ROS wiki 2019). The SRDF file represents information not included in the URDF file, such as collision matrices and initial joint states. The SRDF is used in MOVEit to communicate where the manipulator might collide with the UGV. SRDF files are generated using the MOVEit Setup Assistant, which is included with MOVEit (Görner et al. 2019). After uploading the URDF file, the collision matrix was generated and planning groups made. The completed package and SRDF file was extracted for use in MOVEit.

3.2 Path Planning Functions

MOVEit contains predefined functions that are used to plan the path and move the manipulator (Chitta et al. 2012). There are two types of functions, joint goals and pose goals. With joint goals, each Powertool joint is moved independently of the other joints. This is accomplished by using a joint position function to target joint states. The pose position function accepts a target position for the manipulator relative to the manipulator base and determines the joint positions required to achieve the target. These functions can be accessed using the MOVEit Python Interface after getting the joint or pose states.

One of the requirements established in the previous section is the need for the manipulator to have preconfigured movements that are accessible by pressing a controller button. The ROS package called joy_node publishes joystick controller states. As a result, each joystick button can be linked to

a preconfigured movement. Detailed object inspection is performed via the Panda Powertool using five preconfigured movements: “base,” “rectangle,” “z,” “snake,” and “circle.” These movements are specific to the Panda Powertool and programmed outside of MoveIt. These movements were integrated into MoveIt using Python or C++ interfaces. A Microsoft Xbox One controller is used to initiate preconfigured movements. The controller mappings are shown in Table 1.

Table 1. Preconfigured movements remapped to joystick buttons.

Movement	Description	Button	Type
Base	Return to base position shown in Figure 1	Home	Joint
Rectangle	Move in rectangle around Home position	A	Pose
Z	Move in z pattern passing through Home position	B	Pose
Snake	Move from left to right, moving slightly down at each end until minimum height	X	Pose
Circle	Rotate base joint to minimum and maximum values	Y	Joint

A “home position” motion moves each joint to a predefined position. When the manipulator is in the home position, the end effector, that is, the RGB-D camera, will be positioned directly in front of the robot’s base as shown in Figure 1. This position is used for subsequent motions and is returned to after each motion. The “rectangle” and “z” movements are optimal for quick scans of objects, while the “snake” movement is optimal for detailed scans as it provides a greater number of viewing positions. The full code for the movement functions is listed in Appendix A.

4 Inspection Toggling

Another requirement of this solution is the integration of a toggling system. Specifically, where a detailed mapping task can be initiated and terminated at the operator's discretion using the controller. Typically, packages are launched in a setup sequence, by command line, and run the entire time a robot is executing. This type of program flow is not suitable for the toggling on/off of a detailed mapping node. As a solution, ROS implements a Python application programming interface (API) "roslaunch," which launches a node using a Python script. The node to launch is RTAB-Map's data recorder node, which captures data for postprocessing.

In the implementation, each inspection or detailed mapping should be stored in a separate file, which is not possible using standard RTAB-Map parameters. Therefore, the toggling script must also pass an incrementing filename each time an inspection is initiated. This can be done by concatenating a base name string, number, and file type. Each time a scan occurs, the database file provides a unique name so that each object can be viewed separately.

4.1 Evtest

The ability to write a node that subscribes to joy_node, to get the controller states, seems to be relatively straightforward. However, the roslaunch API functions cannot be called outside of the main function due to dependence on the rospy API. This eliminates the possibility of using a subscriber since it brings the code outside the main function. Due to this limitation, external controller software was used. Specifically, evtest is a Python package that communicates events from an input device to the script (Ubuntu Manuals 2013). This package has several advantages and disadvantages in comparison to joy_node. Evtest can properly launch the RTAB-Map data_recorder without needing to debounce the buttons. However, its functionality leverages the controller event number, which changes on reboot. Thus, checks to determine the controller event number are needed each time the controller is connected to the computer to ensure proper functionality.

4.2 Toggle Controls

When the “Menu” button is pressed on an Xbox One controller, the data recorder node starts saving camera images and transforms into a file called “pandaScan0.db.” The operator sees “Starting RTABMap data_recorder” in the terminal as confirmation of scanning. Once the operator is satisfied with the collection, the “Back” button is pressed to stop the running node. The operator sees “Stopping RTABMap data_recorder” in the terminal as confirmation of shutdown. Additionally, parameters are configured for the next scan. If the operator presses the “Menu” button while already mapping, the operator sees that “RTABMap data_recorder is already running” in the terminal. In contrast, if the operator presses the “Back” button while not mapping, the operator sees that the “RTABMap data_recorder is not running” in the terminal. The full code for the toggling functions is listed in Appendix B.

5 Real-Time Appearance-Based Mapping (RTAB-Map) Data Recorder

RTAB-Map is a simultaneous localization and mapping (SLAM) package that performs loop closure on sensor images, including RGB-D cameras (Labbé and Michaud 2019). By performing loop closures, RTAB-Map is able to accurately recreate a point cloud of the scanned area. Within the RTAB-Map package is the data recorder node. The data recorder node, as the name implies, copies the data from a visual sensor to a database file. The database is used in postprocessing to construct a 3D point cloud of the scanned environment. This technique is most effective for object inspection since another RTAB-Map node is used on the UGV for mapping and localization. For the data recorder node, the "max rate" parameter is set to zero. This allows the data recorder node to copy the camera data as fast as possible to the database. For comparison, the default rate of max rate is typically 1 hz. This high rate of data collection results in an extremely dense 3D point cloud when postprocessed.

5.1 Simulated Object Inspection

After implementing the methodology above, we tested its efficacy in simulation. The TurtleBot3 House world, shown in Figure 2, was used as a test ground for the modified husky. The TurtleBot3 House world was chosen because there are many objects that we can scan. In the simulation, the robot moved around the house and performed scans, on objects of interest, on demand.

5.2 Object Inspection

The first object scanned was the bookshelf in the red box of Figure 2 and was pictured independently in Figure 3. When the robot was positioned in front of the bookshelf, the "Menu" button was pressed, starting the data recorder node. When the confirmation was seen in the console, the "snake" function was performed on the Panda Powertool. Once the Powertool returned to its home position, the "Back" button was pressed, stopping the data recorder node. The second object scanned was the trash can in the blue box of Figure 2 and pictured independently in Figure 4. The same procedure was used to inspect the trash can as was used to scan the bookshelf. After inspections of both objects, the simulation was shut down.

Figure 2. Floorplan of TurtleBot3 House.

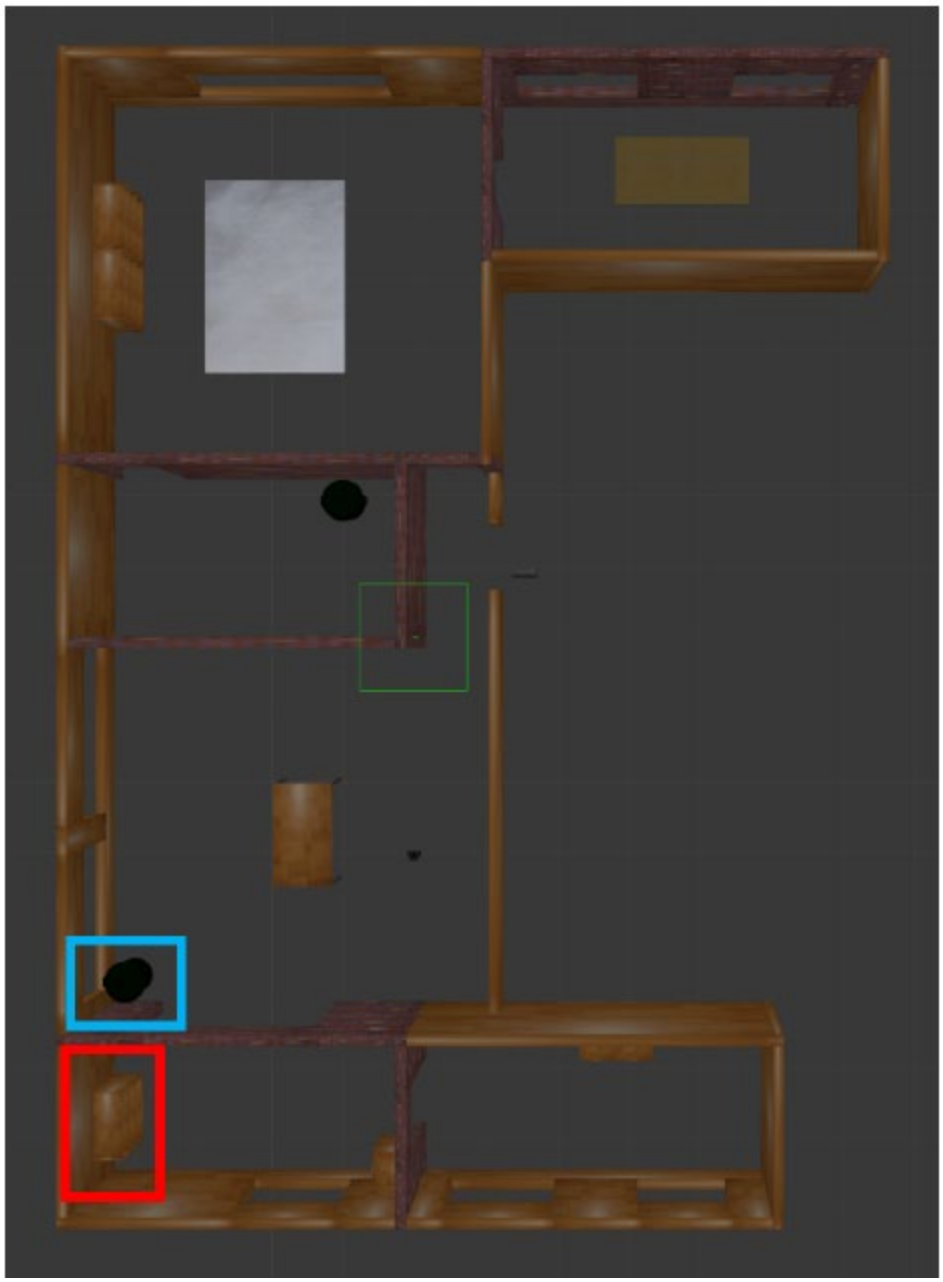


Figure 3. Bookshelf in TurtleBot3 House.

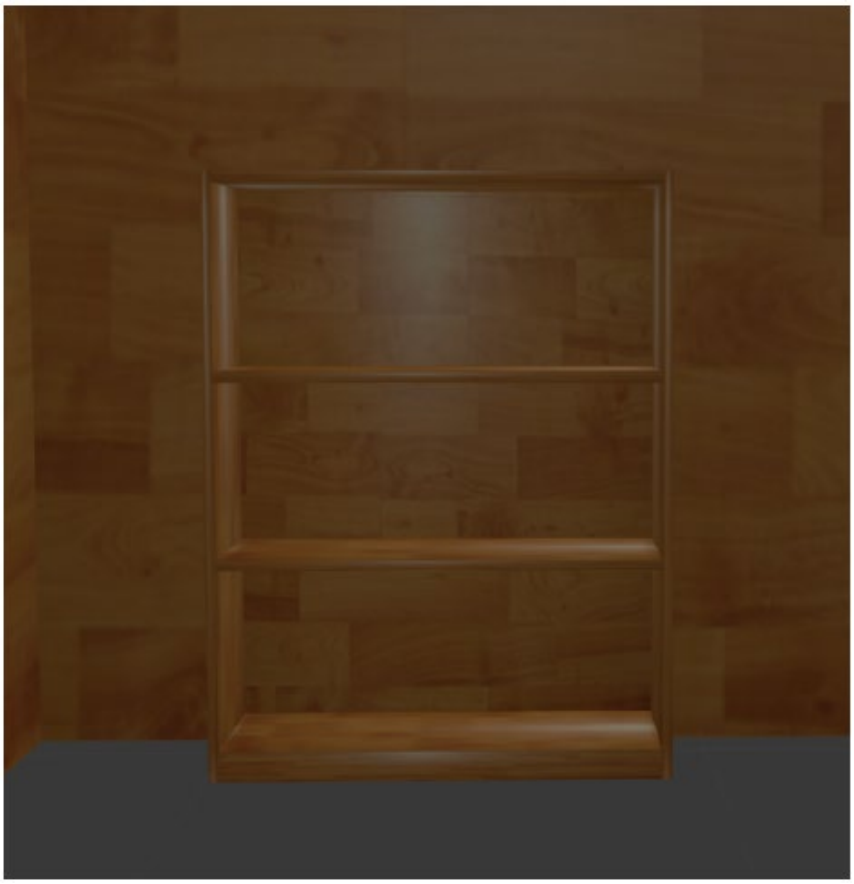


Figure 4. Trash can in TurtleBot3 House.

5.3 Postprocessing

After the object inspection had completed, the operator used the RTAB-Map database viewer to reconstruct the 3D point clouds. The database viewer created visual loop closures using the database file created during inspection. There are many parameters that can be modified when building the point cloud. One particularly useful tool is the meshing parameter, which fills in gaps of the point cloud with colors from the RGB-D camera, creating a more recognizable object in many cases. Below are the suggested steps for creating a meshed point cloud. This procedure was used to generate point clouds with maximum details and minimal error.

1. View—Graph view
2. Edit—Regenerate local grid maps
3. Edit—Detect more loop closures
4. Edit—Refine all loop closure links
5. Edit—Refine all loop closure links
6. Edit—Refine all neighbor links
7. File—Export 3D map
    a. Enable “Cloud Filtering”
    b. Enable “Meshing”
    c. Enable “Regenerate Clouds”
    d. Save

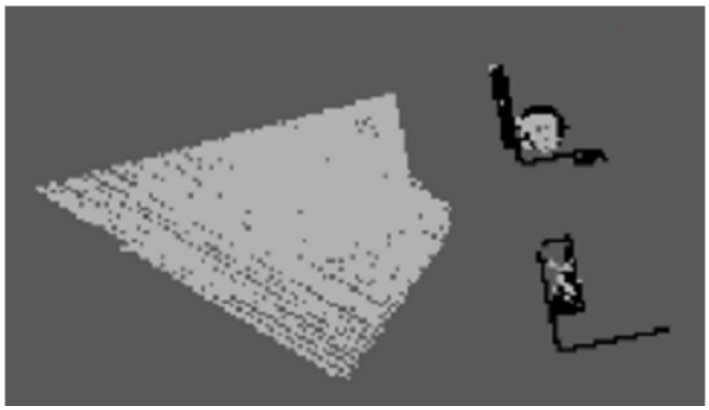
The procedure above creates two products: a 2D occupancy grid and a 3D point cloud (or mesh). The primary benefit of the 2D occupancy grid is that it helps the robot identify obstacles. However, the 2D occupancy grid can also be used to quickly discern whether the scan was successful or needs to be repeated. The 2D occupancy grid, shown in Figure 5, had straight walls and a rectangle corresponding to the bookshelf, which is indicative of a successful scan. The 3D mesh of the bookshelf, shown in Figure 6, looks very similar to the image in Figure 3. The walls are straight with the same pattern of colored rectangular slabs. The frame of the bookshelf also appears to be perpendicular to the wall and the shelves parallel to each other. This result shows that the data collected were sufficient to create a highly accurate meshed point cloud.

Figure 5. Occupancy grid of a bookshelf in 2D.

Figure 6. Mesh of the bookshelf in 3D.

The same postprocessing procedure was applied to the trash can database file to create a 2D occupancy grid and a 3D mesh. The 2D occupancy grid is shown in Figure 7. This occupancy grid looks different than the one depicted in Figure 5 due in part to the camera seeing out of the window and into a neighboring room. Overall, the 2D occupancy grid suggests this was successful as well.

Figure 7. Occupancy grid of a trash can in 2D.



The 3D mesh in Figure 8 appears to align well with the image in Figure 4. The walls surrounding the trash can are seemingly flat and have the correct brick pattern, indicative of a good loop closure. The trash can edges are very clear with no obvious outliers. The handle on the left side of the trash can is also clearly identified in the 3D mesh. Overall, the trash can scan was also successful with clearly identifiable details.

Figure 8. Mesh of the trash can in 3D.

6 Conclusion

The goal of this project was to create a simulated solution for the detailed inspection of objects during mapping. This solution required the use of a robotic manipulator to achieve a greater number of camera viewpoints from which to scan, while the UGV remained still. Additionally, the inspection solution had to be toggled by the operator to create an inspection file for each object encountered. Through the use of the Husky UGV, Panda Powertool, and an RGB-D camera, this goal was achieved. When the operator wishes to inspect a suspicious object, they can toggle the detailed mapping process on. Then, the Panda Powertool was able to manipulate the RGB-D camera about an object using preconfigured movements. The images captured by the RGB-D camera are stored in a database using RTAB-Map. After an inspection event is completed, a 3D mesh was compiled that provided useful details of the object, such as its size, shape, and color rendering.

Based on the simulated results of this report, we recommend pursuing this solution in a real-world environment. A real-world implementation could introduce challenges that are not present in simulation. One such challenge could be unmodeled system dynamics; for example, in simulation, the robotic arm is always stable, whereas in the real world, it is likely to wobble while moving (e.g., due to vibration). An inertial measurement unit could be added to the system to account for the unmodeled system dynamics. Alternatively, RTAB-Map's loop closure and neighbor link refinement methods could mitigate errors caused by unmodeled system dynamics.

Bibliography

Chitta, S., I. Sucan, and S. Cousins. 2012. "Moveit![ros topics]." *IEEE Robotics and Automation Magazine* 19 (1): 18–19.

Clearpath Robotics. 2018. "Husky Simulator." https://github.com/husky/husky_simulator.

Coleman, D., I. A. Sucan, S. Chitta, and N. Correll. 2014. "Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt Case Study." *Journal of Software Engineering for Robotics* 1 (5): 3–16. https://doi.org/10.6092/JOSER_2014_05_01_p3.

Franka Emika. 2022. "franka ros." https://github.com/frankaemika/franka_ros.

Görner, M., R. Haschke, H. Ritter, and J. Zhang. 2019. "MoveIt! Task Constructor for Task-Level Motion Planning." In *IEEE 2019 International Conference on Robotics and Automation (ICRA)*: 190–96. https://doi.org/10.1109/ICRA.2019.8793898

Labbé, M., and F. Michaud. 2019. "RTAB-Map as an Open-Source Lidar and Visual Simultaneous Localization and Mapping Library for Largescale and Long-Term Online Operation." *Journal of Field Robotics* 36 (2): 416–46.

ROS. 2020. "urdf." https://github.com/ros/urdf.

ROS. 2021. "Robot Operating System." https://www.ros.org/.

ROS wiki. 2019. "srdf." https://wiki.ros.org/srdf.

ROS wiki. 2021. "Gazebo." http://wiki.ros.org/gazebo_ros_pkgs.

ROS wiki. 2022. "RTABMap ros." http://wiki.ros.org/rtabmap_ros.

Ubuntu Manuals. 2013. "evtest." https://manpages.ubuntu.com/manpages/focal/man1/evtest.1.html.

Appendix A: Movement Functions

```
def go_to_base_joint_state(self):
  # Copy class variables to local variables to make the web
 tutorials more clear.
  # In practice, you should use the class variables directly
 unless you have a good
  # reason not to.
  move_group = self.move_group

  ## BEGIN_SUB_TUTORIAL plan_to_pose ##
  ## Planning to a Pose Goal ##
  ^^^^^^^^^^^^^^^^^^^^^^^^
  ## We can plan a motion for this group to a desired pose
  for the
  ## end-effector:
  joint_goal = move_group.get_current_joint_values()
        joint_goal[0]    = pi/2
        joint_goal[1]    = pi/2
        joint_goal[2]    = -2.0471461344852866
        joint_goal[3]    = -2.4479860982865604
        joint_goal[4]    = -1.951829599344788
        joint_goal[5]    = 1.862655594996444
        joint_goal[6]    = 0.9390001998549142

  # The go command can be called with joint values, poses,
 or without any
  # parameters if you have already set the pose or joint
 target for the group
  move_group.go(joint_goal, wait=True)

  # Calling ``stop()`` ensures that there is no residual
 movement
  move_group.stop() ## END_SUB_TUTO-

  RIAL


def rectangle(self, scale=1):
  # Copy class variables to local variables to make
the the web tutorials more clear.
# In practice, you should use the class variables
directly unless you have a good # rea-
 son not to.
 move_group = self.move_group
   ## BEGIN_SUB_TUTORIAL plan_cartesian_path ##
   ## Cartesian Paths ##
   ^^^^^^^^^^^^^^^^
   ## You can plan a Cartesian path directly by
 specifying a list of waypoints
   ## for the end-effector to go through. If executing interac-
    tively in a
```

```
    ## Python shell, set scale = 1.0. ##
    waypoints = []

    wpose = move_group.get_current_pose().pose wpose.position.y
    -= 0.4 # First move left in (y) wpose.position.z += 0.35 #
    and move up in (z) waypoints.append(copy.deepcopy(wpose))

    wpose.position.y += 0.8 # First move right in (y) way-
    points.append(copy.deepcopy(wpose))

    wpose.position.z -= 0.7 # and down in (z) waypoints.ap-
    pend(copy.deepcopy(wpose))

    wpose.position.y -= 0.8 # Third move sideways (y) way-
    points.append(copy.deepcopy(wpose))

    wpose.position.z += 0.7 # and move up in (z) waypoints.ap-
    pend(copy.deepcopy(wpose))

    # We want the Cartesian path to be interpolated at a resolu-
  tion of 1 cm
    # which is why we will specify 0.01 as the eef_step in Carte-
    sian
    # translation. We will disable the jump threshold by setting
  it to 0.0,
    # ignoring the check for infeasible jumps in joint space,
  which is sufficient
    # for this tutorial.
    (plan, fraction) = move_group.
 compute_cartesian_path(

waypoints to follow eef_step jump_threshold

waypoints, #

0.01,    #

0.0)    #


    move_group.execute(plan, wait=True)

 def cartesianZ(self, scale=1):
    # Copy class variables to local variables to make the web
  tutorials more clear.
    # In practice, you should use the class variables di-
  rectly unless you have a good
    # reason not to.
    move_group = self.move_group

    ## BEGIN_SUB_TUTORIAL plan_cartesian_path ##
    ## Cartesian Paths ##
    ^^^^^^^^^^^^^^^^
    ## You can plan a Cartesian path directly by specifying a
  list of waypoints
```

```
   ## for the end-effector to go through. If executing interac-
    tively in a
   ## Python shell, set scale = 1.0. ##
   waypoints = []

   wpose = move_group.get_current_pose().pose

   wpose.position.y -= 0.4 # First move left in (y) wpose.posi-
   tion.z += 0.35 # and move up in (z) waypoints.append(copy.deep-
   copy(wpose))

   wpose.position.y += 0.8 # First move right in (y) waypoints.ap-
   pend(copy.deepcopy(wpose))

   wpose.position.y -= 0.4 # First move left in (y) wpose.posi-
   tion.z -= 0.35 # and move up in (z) waypoints.append(copy.deep-
   copy(wpose))

   wpose.position.y -= 0.4 # First move left in (y) wpose.posi-
   tion.z -= 0.35 # and move up in (z)
waypoints.append(copy.deepcopy(wpose))
   wpose.position.y += 0.8 # Third move sideways (y) way-
   points.append(copy.deepcopy(wpose))

   # We want the Cartesian path to be interpolated at a reso-
 lution of 1 cm
   # which is why we will specify 0.01 as the eef_step in Car-
   tesian
   # translation. We will disable the jump threshold by set-
 ting it to 0.0,
   # ignoring the check for infeasible jumps in joint space,
 which is sufficient
   # for this tutorial.
   (plan, fraction) = move_group. compute_carte-
 sian_path(

waypoints to follow eef_step jump_threshold



waypoints, #

0.01,    #

0.0)    #

   move_group.execute(plan, wait=True)

def cartesianSnake(self, scale=1):
   # Copy class variables to local variables to make the web
 tutorials more clear.
   # In practice, you should use the class variables directly
 unless you have a good
   # reason not to.
```

```
  move_group = self.move_group

  ## BEGIN_SUB_TUTORIAL plan_cartesian_path ##
  ## Cartesian Paths
  ## ^^^^^^^^^^^^^^^
  ## You can plan a Cartesian path directly by
 specifying a list of waypoints
  ## for the end-effector to go through. If executing interac-
   tively in a
  ## Python shell, set scale = 1.0. ##
  waypoints = []
  maxYDisp = 0.8
  reps = 0
wpose = move_group.get_current_pose().pose
wpose.position.y -= 0.4
 wpose.position.z += 0.35 waypoints.append(copy.deepcopy(wpose))

 while (reps < 4): wpose.position.y += maxYDisp
      waypoints.append(copy.deepcopy(wpose))
      wpose.position.z -= 0.14 waypoints.ap-
      pend(copy.deepcopy(wpose)) maxYDisp =
      maxYDisp * -1
      reps += 1
 else:
      wpose.position.y += maxYDisp way-
      points.append(copy.deepcopy(wpose))
      (plan, fraction) = move_group.
compute_cartesian_path(


waypoints,
                                                                  #
waypoints to follow
0.01,    #
eef_step jump_threshold


0.0)    #

       move_group.execute(plan, wait=True)

def circleSweep(self):
  # Copy class variables to local variables to make the web
 tutorials more clear.
  # In practice, you should use the class variables di-
 rectly unless you have a good
  # reason not to.
  move_group = self.move_group

  ## BEGIN_SUB_TUTORIAL plan_to_joint_state ##
  ## Planning to a Joint Goal ##
  ^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
   ## The Panda's zero configuration is at a ' singularity
 <https://www.quora.com/Robotics-What-is- meant-by-kinematic-
 singularity>'_ so the first
   ## thing we want to do is move it to a slightly better
 configuration.
# We can get the joint values from the group and
adjust some of the values:
  joint_goal = move_group.get_current_joint_values() base_joint =
  joint_goal[0]
  joint_goal[0] = -7*pi/8

  # The go command can be called with joint values, poses,
or without any
  # parameters if you have already set the pose or joint
target for the group
  move_group.go(joint_goal, wait=True)

  # Calling ``stop()`` ensures that there is no residual
movement
  move_group.stop()

  joint_goal[0] = 7*pi/8

  # The go command can be called with joint values,
poses, or without any
  # parameters if you have already set the pose or joint
target for the group
  move_group.go(joint_goal, wait=True)

  # Calling ``stop()`` ensures that there is no residual
movement
  move_group.stop()

  joint_goal[0] = base_joint

  # The go command can be called with joint values, poses,
or without any
  # parameters if you have already set the pose or joint
target for the group
  move_group.go(joint_goal, wait=True)

  # Calling ``stop()`` ensures that there is no residual
movement
  move_group.stop()
## END_SUB_TUTORIAL
```

Appendix B: Toggling Code

```
#!/usr/bin/env python
#import evdev
from evdev import InputDevice, categorize, ecodes im-
port roslaunch
import rospy


rospy.init_node('en_Mapping', anonymous=True) uuid =


roslaunch.rlutil.get_or_generate_uuid(None,
   False) roslaunch.configure_log-
ging(uuid)

fileNumber = 0
cli_args    =    ['/home/mowles/testMin_ws/src/rtabmap_ros/
   launch/rtabmapToggle/panda_data_recorder.launch','  out-
   put_path:=pandaScan' + str(fileNumber) + '.db']
roslaunch_args      =      cli_args[1:]
roslaunch_file = [(roslaunch.rlutil.
   resolve_launch_arguments(cli_args)[0],
   roslaunch_args)]

parent = roslaunch.parent.ROSLaunchParent(uuid, roslaunch_file)
#creates object controller
controller = InputDevice('/dev/input/event5')

bakBtn = 314
strtBtn = 315
toggle = False
#Toggle rtabmap data_recorder on and off.
for event in controller.read_loop():
 #Buttons
 if event.type == ecodes.EV_KEY:
  #Start and stop data_recorder when button presser if
     event.value == 1:
      if (event.code == bakBtn) and toggle: print("Stop-
       ping RTABMap data_recorder") parent.shutdown()
fileNumber += 1
          cli_args = ['/home/mowles/testMin_ws/src/
rtabmap_ros/launch/rtabmapToggle/panda_data_recorder
.launch','output_path:=pandaScan' + str(fileNumber)
+ '.db']
    roslaunch_args = cli_args[1:] roslaunch_file
    = [(roslaunch.rlutil.
resolve_launch_arguments(cli_args)[0],
roslaunch_args)]
    parent = roslaunch.parent.ROSLaunchParent(uuid,
roslaunch_file)
    toggle = False
   elif (event.code == strtBtn) and not toggle:
    print("Starting RTABMap data_recorder") parent.start()
```

```
      rospy.loginfo("started")
      toggle = True
    elif (event.code == bakBtn) and not toggle:
      print("RTABMap data_recorder is not running")
    elif (event.code == strtBtn) and toggle:
print("RTABMap data_recorder is already running")
```

Abbreviations

API	Application programming interface
DoF	Degrees of freedom
NDT	Nondestructive testing
RGB-D	Red-green-blue and depth
ROS	Robot Operating System
RTAB-Map	Real-Time Appearance-Based Mapping
SLAM	Simultaneous localization and mapping
SRDF	Semantic Robot Description Format
SWaP-C	Size, weight, power, and cost
UGV	Unmanned ground vehicle
URDF	Unified Robot Description Format

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
November 2023	Final Technical Report (TR)	FY19	FY22

4. TITLE AND SUBTITLE

Leveraging MOVEit for Object Inspection in Simulation

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT
		29785D
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**
497149	8	

6. AUTHOR(S)

Nathan Dodson, Mike Paquette, and Garry Glaspell

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

US Army Engineer Research and Development Center (ERDC)
Geospatial Research Laboratory (GRL)
7701 Telegraph Road
Alexandria, VA 22315-3864

8. PERFORMING ORGANIZATION REPORT NUMBER

ERDC/GRL TR-23-5

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Headquarters, US Army Corps of Engineers Washington, DC 20314-1000	HQUSACE	

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Herein we evaluate using a robotic arm with an attached camera to investigate objects of interest in simulation. Specifically, a Husky unmanned ground vehicle with a Panda Powertool was used in the simulation. The code enabled an operator to initiate a preconfigured set of motions when an object of interest was identified. The scan was stored in a database file that was used to generate a 3D mesh of the scanned object. The report describes both setting up the simulation and the code used to scan objects of interest.

15. SUBJECT TERMS

Computer programs--Evaluation; Hydraulic structures--Inspection; Robots--Control systems; Robots--Kinematics

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
Unclassified	Unclassified	Unclassified	SAR	33

19a. NAME OF RESPONSIBLE PERSON	19b. TELEPHONE NUMBER (include area code)