UNIVERSITY OF AMSTERDAM

RESEARCH PROJECT

Technical Report

July 1, 2024

Lecturer:
Arnoud Visser

Student:
Milan La Riviere, Mark Honkoop,
Gideon Pol
13379275, 13317466, 15332837

Course:
Master Computer Science
Research Project

Course code:
XM_0129

Contents

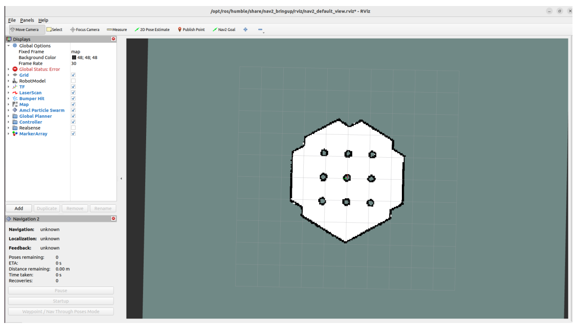
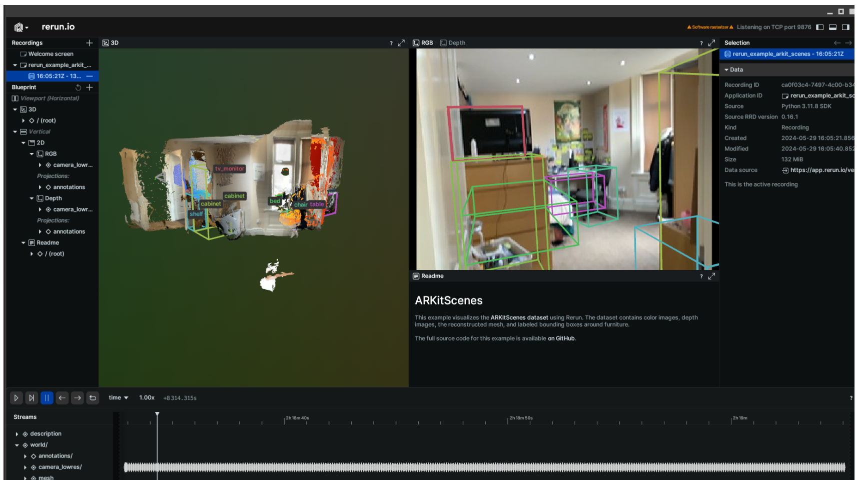


Figure 1: RVIZ interface



ARKitScenes

This example visualizes the **ARKitScenes dataset** using Rerun. The dataset contains color images, depth images, the reconstructed mesh, and labeled bounding boxes around furniture.

The full source code for this example is available **on GitHub**.

Figure 2: Rerun interface

1 Introduction

This project aims to create a ROS 2 node that logs various data types from ROS to Rerun. This node is called Swanky henceforth. The implementation can be found at `https://github.com/MilanLR/ROS2-rerun-integration`.

1.1 ROS 2

The Robot Operating System (ROS) is a set of software libraries and tools for building robot applications [5]. ROS 2 is an improved, and newer version of ROS which is better suited for today's robots. The default visualizer for ROS 2 is often RVIZ (with an interface like Figure 1. Rerun is a newer, more modern implementation, more suited for the use cases of today.

1.2 Rerun

Rerun is an SDK, time-series database, and visualizer for temporal and multi-modal data. It's used in fields like robotics, spatial computing and 2D/3D simulation to verify, debug, and visualize [3]. Rerun looks like Figure 2 when launching one of the templates.

1.3 Why combining Rerun and ROS 2

ROS 2 is used for many robots[2] because of its flexibility in sensors. Rerun is used for debugging and logging of robots and their sensors. It makes sense to combine these two programs to visualize and debug the data coming from ROS 2. This has not been done, however, except for specific use cases [6]. A generic node would make the connections between ROS 2 and Rerun effortless, as it would just require to spin up another ROS 2 node.

1.4 How to combine Rerun and ROS 2

A connection between ROS 2 and Rerun can be implemented in different ways, some building upon others to make it more efficient or easier to use. Research yielded the following primary methods for implementing this bridge: [1]:

- **Generic Rerun node** Given that Rerun and ROS do not necessarily operate on the same data formats or types, a *bridge* between Rerun and ROS 2 will be necessary to close the gap between said differing formats. This bridge should be generic and therefore configurable, meaning the bridge should not be specifically tailored to a particular set of sensors and/or platforms, but can be adapted with minimal effort to be compatible with a new device.

  In ROS 2, modules communicate through *topics*, which serve as endpoints for sensors to submit their data. Ideally, the bridge should automatically detect the available topics and convert the message of such a topic from the topic's ROS 2 type into a Rerun type, thereby also submitting the data to a running Rerun client.

- **generic ROSbag-loader** ROSbag is a tool for working with *bags*. A bag in ROS 2 is a file format that allows the storage of submitted topic data, which can be loaded to retrieve data from a previous point in time. Visualizing such a bag in Rerun could prove valuable as it would eliminate the requirement of having Rerun running at the same time the data is generated. Instead, bags would allow the gathering of data out in the field while still being able to inspect the data's history in Rerun.

- **Viewer-side ROS message handlers** Building on top of a more generic Rerun node, the next step would be to do more handling of ROS messages directly in Rerun. This would allow relaying all messages fully generically into the store without ever deserializing the payload, vastly simplifying the bridge code.

- **Viewer-side plugin for receiving messages** A viewer-plugin that acts like a ROS node could in theory inject messages directly into Rerun without needing to go through a separate log step, removing one more copy from the pipeline.

In this project, a generic ROS 2 bridge is implemented for Rerun. This is a ROS 2 node that subscribes on all (or a selected few) topics and logs the incoming data to Rerun. This method appears to be the best first step towards integration and would provide the best baseline understanding of ROS and its data types. This method was chosen over the generic ROSbag-loader approach as bags do not deal with live data and are therefore not as useful for an initial implementation. Secondly, a viewer-side plugin for receiving messages can only work when ROS 2 and Rerun run on the same machine. As such, it cannot be used on the Mini Pupper 2 and turtlebot, thereby also eliminating this option for initial implementation.

2 Related Literature

This project was founded after Rerun Released an example of how to use it with ROS 2 [1]. While this example was working for a simulated Turtle-bot 3, it admitted that there was a "non-trivial amount of code" to process each ROS 2 message and log them to Rerun. At that time there was a ROS 1 bridge that acted as a ROS 1 node, subscribed to supported topics, and transmitted data from those topics to Rerun[2]. While working on this project, the ROS bridge was ported to ROS 2[3]. It also auto-subscribes to all the supported topics and logs them to Rerun. It does, however, require the user to manually define how the components in the URDF tree should be logged to Rerun.

[1]https://github.com/rerun-io/cpp-example-ros-bridge
[2]https://github.com/rerun-io/cpp-example-ros-bridge/tree/main
[3]https://github.com/rerun-io/cpp-example-ros2-bridge/tree/main

2.1 Related Rerun GitHub Issues

The Rerun code on github, has a few issues related to it. These are from people that use Rerun, and feel like it is missing functionality or that there are bugs that are making the code not optimal.

2.1.1 Support for LaserScan projection models

One issue, mentioned in this issue, shows that there is no Rerun support for LaserScan projections. This data is available in ROS 2, and having Rerun support would increase the functionality of Rerun when working with ROS 2 and Rerun. While this project did not fix this issue, the LaserScan data type is converted to Rerun LineStrips2D in the implementation that comes with this paper. This way, the LaserScan data can still be logged to Rerun.

2.1.2 Support for Derived Transforms

Another problem that was mentioned in the Native Ros support issue is the support for derived transforms. The robot components are described in a URDF component tree, the same tree that must be used to log each component to Rerun. However, this tree might differ from the data tree that should be logged, which in this case is the tf2 tree that describes the transforms of each component.

These tf2 “paths” have to be first converted to the correct URDF component path, before they can be logged correctly. The Rerun issue is looking to support logging angles relative to another component. This would, for example, make it possible to log the angle of a lower arm relative to an upper arm. For Swanky to correctly log URDF files updated with the joint angles, it has to support these “derived” transforms, however, this support is build into the ROS node, and not into Rerun itself.

2.1.3 Support for more 2D and 3D primitives

The last relative issue is Support for more 2D and 3D primitives. When implementing SLAM, the same issue was encountered in this project. It was fixed with custom code in the ROS 2 node. But this does not resolve the problem for Rerun.

3 Method

3.1 Setup

The three major objectives of the generic node (Swanky) are as following:

1. When turned on, the generic node will find each open endpoint, subscribe to it, and transform the data such that it can be logged to Rerun. Where possible direct conversions will be performed if data types correspond one-to-one between the two platforms. However, where there is no such direct correspondence (for example, depth images and regular images have the same ROS data type but different Rerun data types),

   This will have some guesses, since depth images and images, for example, have the same ROS data type, but different Rerun data types. So, in that case, it is not possible to know for sure which data type to give to Rerun.

2. The node is easily configurable with a JSON file or Python dictionary that allows control over the endpoints that the node must subscribe to.

3. Given that the list of topics, and thereby required libraries, can become extensive, the Python libraries must be imported dynamically based on the subscribed topics. It should not be necessary to install the libraries for all topics to be able to run the node.

3.2 Robots

To show that Swanky not only works on isolated sensors but can handle larger configurations, it has been tested on two different robots, The Mini Pupper 2 and the Turtle-bot 3. When using a separate platform with its own operating system, the ROS node and the Rerun server must run on different machines. Here, ROS runs on the robot while a PC hosts a Rerun session for debugging. Rerun has native support for logging over the internet, which can be configured using an IP address and port, allowing for a seamless integration that barely requires extra effort to setup.

3.2.1 Mini Pupper 2

Given that tf2 transformations are paramount to the ROS system, correctly implementing tf2 transformations in Swanky is a must. As such, testing on a robot equipped with joints is by extension a requirement. Therefore, the Mini Pupper 2 by MangDang (see Figure 3) has been chosen as its legs are very transformation-heavy and can be used to test the joint logging. Additionally, the Pupper is equipped with a 360-degree LiDAR and an IMU sensor, of which the LiDAR is an especially interesting sensor to log to Rerun. Finally, there is a URDF file available for the Pupper. URDF files contain information about the robot's visualization (3D meshes and textures), as well as a description of the transformation hierarchy of its physical components. They are often used in robotics to visualize robots, so this is something that should be implemented in Swanky as well.

The Mini Pupper 2 runs Ubuntu 22.04 and ROS 2 Humble. It has a battery that lasts about 30 to 40 minutes if the Pupper is not walking. The Pupper can be turned on with a Raspberry Pi 4 power supply, but in that case, the servo motors are not working. Additionally, the battery cannot be charged while it is supplying power to the Pupper. This slows down development significantly as whenever the battery needs to be charged, it is not possible to perform tests that require the servo motors.

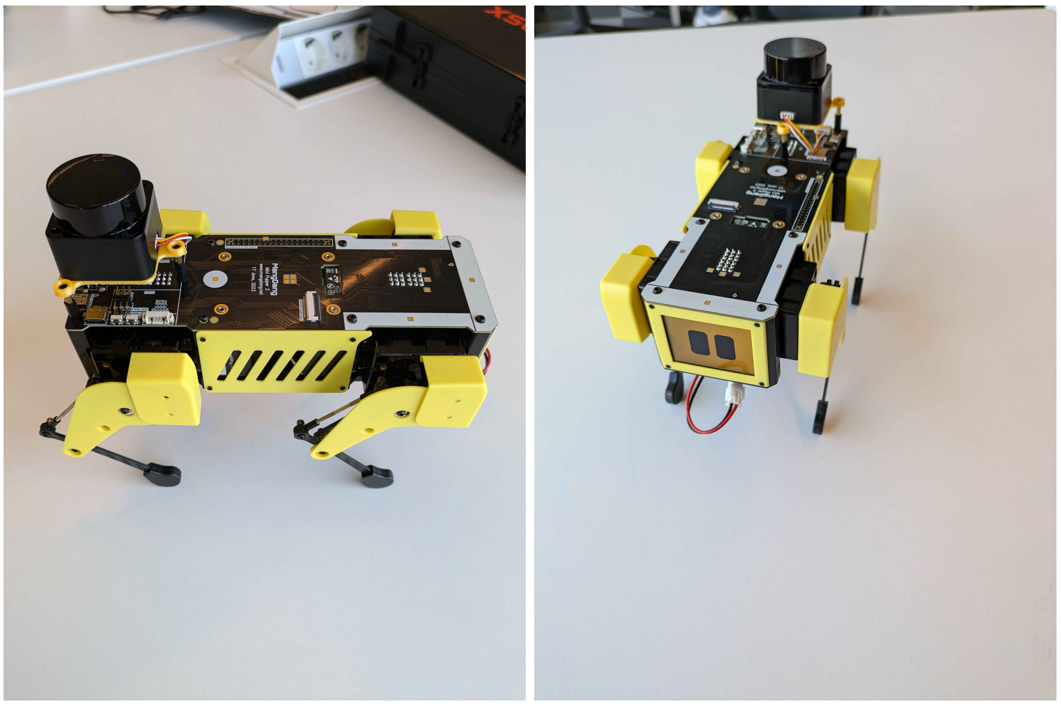


Figure 3: Mini Pupper 2 by MangDang

3.2.2 Turtle-bot

Unlike the Pupper robot, the turtle-bot features no complicated joints that require transformations. It's a rather simple, static robot featuring 2 separately controllable wheels. The drivers of these wheels also contain odometers, meaning their rotation can be tracked to measure the movement of the robot. Like the Pupper, the turtle-bot has a 360-degree LiDAR sensor on top of its frame and is capable of driving around rather quickly on battery power. The turtle-bot is controlled by a Raspberry Pi 4, which was configured to run Ubuntu server 22.04 as per the ROS 2 Humble recommendation.

Because of its simplicity, the turtle-bot is a widely used and supported platform in the field of robotics. This also goes for ROS, for which there already exist nodes to control and read the turtle-bot. The default nodes can log the odometry, LiDAR data, and a predefined URDF, although there is also support for SLAM with extra packages. Coincidentally, in these examples also exists a node capable of logging its most important sensory data to Rerun, making it a good starting point for the generic node.

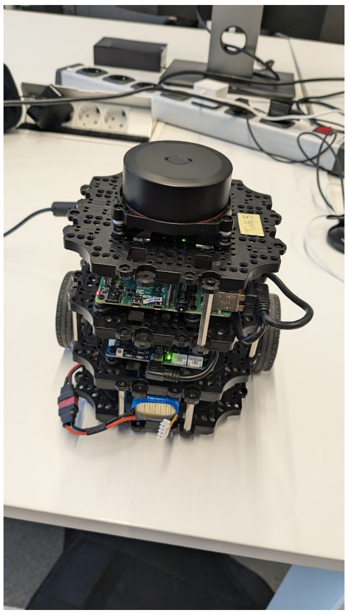


Figure 4: Turtle-bot

3.2.3 Camera's

Next to the sensors on the mini Pupper and turtle-bot, a few camera sensors have been added as well to Swanky. The first being a generic camera, and the second being a depth camera. The generic camera only outputs an image, which needs to be decoded using the right encryption. A depth camera (see Figure 5) outputs a depth image, and occasionally an IR image, point cloud and normal image — depending on the camera. The normal image works the same as with the normal camera. The Depth image and IR image can be logged as Depth Images in Rerun (with the proper decoding). The point cloud can be logged to Rerun as Points3D. The point cloud can however also have colours, which have been considered too.

4 Result

Swanky bridges the gap between ROS 2 and Rerun. Its purpose is to make it easy to visualize different sensors in Rerun, requiring minimal modification of the code to introduce additional sensory inputs. Swanky subscribes to a list of ROS topics, either supplied by the user or to all available topics. Given support for the topic's data type has been implemented in Swanky, it will

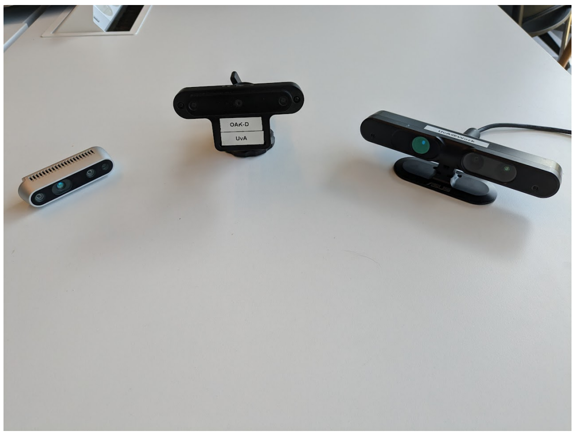


Figure 5: Depth cameras

automatically perform the necessary transforms to log the data to Rerun. The currently supported types are listed down below. Additionally, if a URDF file has been provided to Swanky, the node can also visualize the joint movements.

- LaserScan
- JointTrajectory
- Odometry
- PointCloud2
- Image (Can log a normal image or a depth image to Rerun, depending on the encoding of the image)
- CompressedImage
- OccupancyGrid

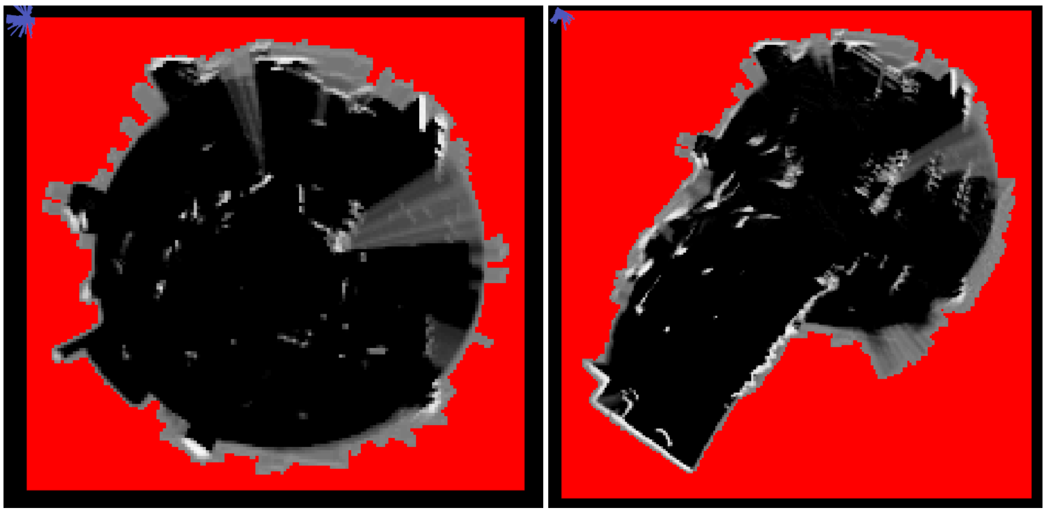


Figure 6: Mini Pupper 2 running SLAM before (left) and after (right) walking next to a wall

4.1 URDF files

As mentioned in subsubsection 3.2.1, URDF files describe what a robot and its components look like using transformations and meshes. These transformations include the position and orientation of the joints relative to other components, making URDF files perfect for visualizing the joint data of robots. There already exists a Rerun example to log a URDF file[4]. However, this example was logged as a recording and therefore had to be modified so that it could work with live data, and so that the logged meshes could be updated with live motor rotations. To be able to link joints to their URDF path, a lookup table had to be generated as shown in Listing 1. This makes it possible to map tf2 transformations to the correct URDF joint. Figure 7 shows a visualization of the Mini Pupper in Rerun using a URDF file, which is done through the Rerun node.

Alternatively, the node can also subscribe to a callback from the ROS Robot State Publisher[4], which is capable of publishing a URDF modified with joint transformations in real-time.

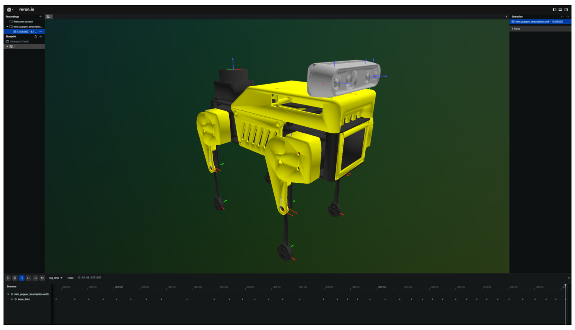


Figure 7: Pupper URDF file loaded in Rerun

```python
def get_joint_path_map(urdf_logger):
    def joint_path(joint):
        root_name = urdf_logger.urdf.get_root()
        joint_names = urdf_logger.urdf.get_chain(root_name, joint.child)[
            0::2
        ]
        return (
            joint.name,
            urdf_logger.add_entity_path_prefix("/".join(joint_names)),
            joint,
        )

    return {
        joint_name: (joint_path, joint)
        for (joint_name, joint_path, joint) in [
            joint_path(joint) for joint in urdf_logger.urdf.joints
        ]
    }
```

Listing 1: Creating the joint path map. This maps a joint name to the joint path and the object containing the joint information in Python.

[4]https://github.com/rerun-io/rerun-loader-python-example-urdf

4.2 Robots with joints

The ROS 2 node works for the Mini Pupper 2. The rotations of a robot component should be visualized relative to the parent component, not to a single absolute point in space. Swanky uses tf2 and a URDF file to correctly visualize the angles of all the joints. Any robot that has a URDF file available and publishes its joint positions using tf2 should be compatible with Swanky. The only requirement is that tf2 publishes the joint angles with the same parent and child names as defined in the URDF file. Figure 8 shows a URDF visualization updated with joint angle data.

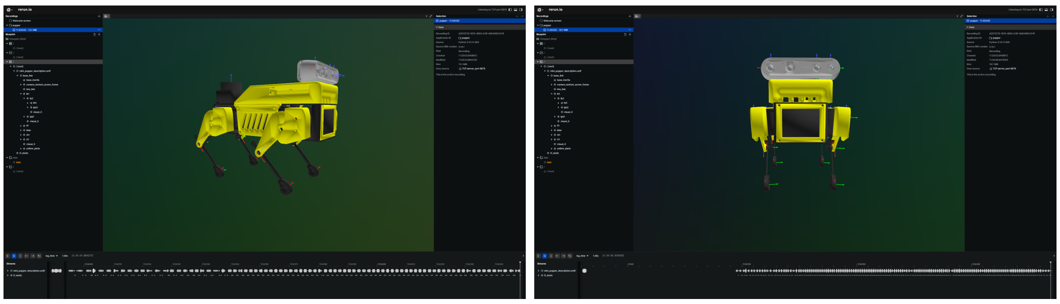


Figure 8: Mini Pupper visualized in Rerun using a URDF file updated with the joint angle data.

As mentioned in subsubsection 2.1.2, the tf2 transformations of the robot's components have to get linked to the correct component defined in the URDF file. This is done by modifying Rerun's example on how to load URDF files. A lookup table is generated to map a component's name to their absolute path defined in the URDF file. This was one of the possible solutions mentioned in this Rerun issue. This lookup table makes it possible to map a tf2 transform to the correct URDF component, after which logging in Rerun is simple.

4.3 LiDAR

Any LiDAR that publishes its measurements using the standard ROS 2 scan data struct (LaserScan) is compatible with the Rerun node. The LaserScan data format is not supported by Rerun, meaning Swanky must convert the data to a suitable representation. Figure 9 shows two different visualization methods Swanky supports for LiDAR. The left image only logs the points where the LiDAR detects an object, with objects closer by getting a redder colour. The right image shows line segments from the centre of the LiDAR to the detected objects, which can make it clear what areas of observation are blocked by objects.

The LiDAR data is supplied through the topic as an angular range specified by a min and a max, along with measured distances at each angle (where the delta is determined by the range and number of measurements.) Through some basic trigonometry, these angles and distances can be converted to 2D points, which can easily logged to Rerun (as shown in Listing 2.)

```python
def laserscan_callback(msg, topic_name):
    num_samples = len(msg.ranges)
    lin_space = np.linspace(msg.angle_min, msg.angle_max,
        num_samples)

    x_values_1 = np.cos(lin_space) * msg.range_min
    y_values_1 = np.sin(lin_space) * msg.range_min
    x_values_2 = np.cos(lin_space) * msg.ranges
    y_values_2 = np.sin(lin_space) * msg.ranges

    match self.lidar_visualization_option:
        case LidarVisualizationOption.Lines:
            for i in range(num_samples):
                start_point = (x_values_1[i], y_values_1[i])
                end_point = (x_values_2[i], y_values_2[i])
                log_line(start_point, end_point)
        case LidarVisualizationOption.Colour:
            for i in range(num_samples):
                end_point = (x_values_2[i], y_values_2[i])
                log_point(end_point)
```

Listing 2: LaserScan Callback Function

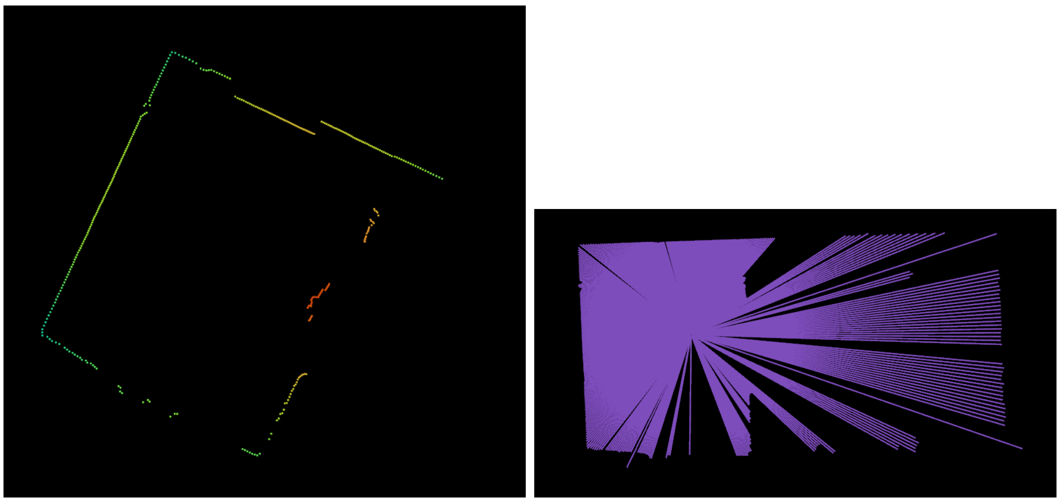


Figure 9: Visualization of the Pupper's LiDAR in Rerun with different visualization methods.

4.4 SLAM

As both robots featured in this project grant access to LiDAR and odometry data, both are capable of performing so-called Simultaneous Localization and Mapping (SLAM). SLAM aims to create and update a mapping of an agent's unknown environment while also keeping track of the agent's position. SLAM is a readily solved problem in ROS 2, which provides a "cartographer" node that automatically subscribes to the necessary topics and outputs topics itself with the computed mappings.

The primary output of the cartographer node is an OccupancyGrid. This occupancy grid is the SLAM's representation of the environment, mapping out the occupied and free spaces in cells. The values of these cells can range from -1 to 100, where -1 represents unknown space, indicating that the cell's status has not been observed or determined yet. 0 represents free space, suggesting that the cell is navigable and there are no obstacles and 1 to 100 represent varying degrees of occupancy, with higher values indicating a higher probability that the cell is occupied by an obstacle.

Such a grid can easily be converted to an image by scaling 0-100 to 0-255 for every color channel and picking a separate color to represent -1. Figure 6 shows an image representation of an occupancy grid as logged in Rerun.

4.4.1 Image Processing

Representing the SLAM data as an image comes with the added benefit of being able to use image processing libraries such as OpenCV to analyse or transform the data. Figure 11 shows the application of a probabilistic Hough line transform on the SLAM data. Such a transformation is capable of detecting straight lines in an image and could thus be used as a means of extracting the outlines of rooms. One problem with probabilistic Hough, is that it returns many segments, many of which may be redundant or overlap others. This redundancy can clutter the output with multiple segments representing the same line, making it difficult to identify features in the SLAM data. To minimize this clutter, the algorithm in Listing 3 could be used to reduce the number of segments:

```
for every line:
    if line goes right to left:
        line = (line.end, line.start) # Swap start and end

lines.sort(by=length)

def lines_mergable(line1, line2):
    return
      (
        abs(line1.slope - line2.slope) < \gamma or
        # Treating line1 as being infinite
        dst(line2.start, line_1) < ϵ and dst(line2.end, line_1) < ϵ
      ) and
      (start and endpoints of line1 and line2 lie close to
         eachother)

def merge_lines(line1, line2):
    points = [line1.start, line2.start, line1.end, line2.end].sort(
       by=(x,y))
    return (points[0], points[-1])

for every line1:
  for every line2 that is not line1:
    if lines_mergable(line1, line2) then:
        merge_lines(line1, line2)
        lines.remove(line2)
```

Listing 3: Pseudocode for SLAM with Hough transformations

- All lines are normalized to go from left to right.
- The lines are sorted by length as longer lines are likely of larger importance and less prone to noise.
- Every possible line pair is tested for mergeability, as determined by the slope of both lines and the proximity of their extremities. Alternatively, the condition for the slope can also

be satisfied if both the start and end of the smaller line (line2) lie close to the longer line (line1), which proved useful for filtering noisy smaller segments.

- The lines are merged by performing a cartesian sort (the reason normalization was necessary) and picking the start and end points from the resulting list.

The result of this filtering operation can be seen in Figure 10.

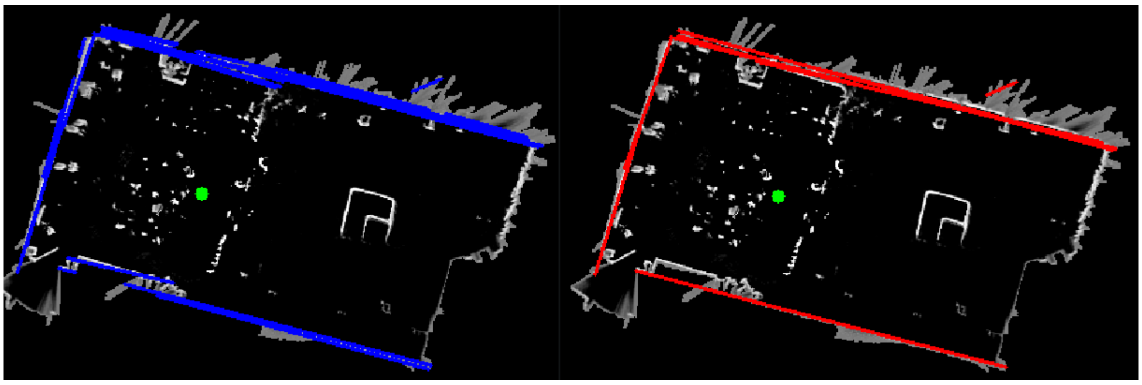


Figure 10: Probabilistic Hough line transform with and without line filtering.

This derivation can either be visualized in 2D as lines on the grid's image representation or alternatively as a 3D mesh, as shown in Figure 11.

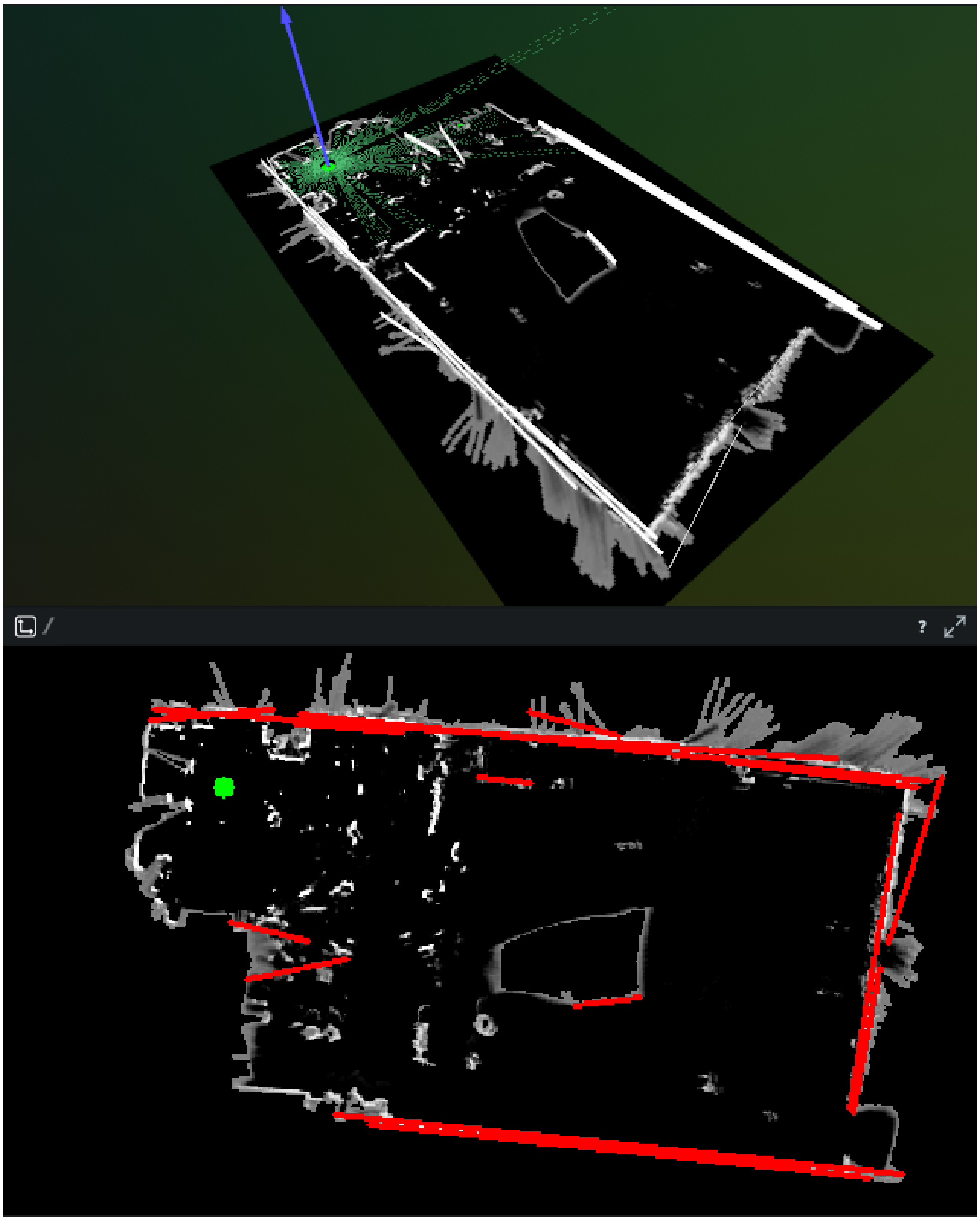


Figure 11: 3D mesh constructed from Hough lines.

4.5 Image and CompressedImage

The Image and CompressedImage both emit the same ROS type, be it with different encodings. These encodings form the primary challenge when correctly logging the image to Rerun, as each encoding works differently and needs a distinct decoding function to work properly. For the scope of this project, a select number of encodings have been implemented. The ability to add encodings to the implementation is presented, with a clear Error when outputting one that is not implemented yet Listing 4.

When an encoding type is found that is normally used for depth images — like **16UC1**,

8UC1, or **32FC1** — the image is logged to Rerun as a depth image.
The resulting images in Rerun look like Figure 12.

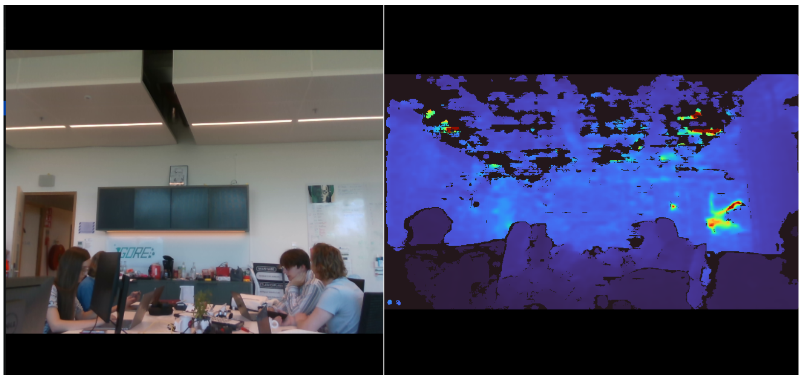


Figure 12: Rerun visualization of image (left) and depth image (right)

```
function image_callback(img, topic_name):
    set_ros_time()

    if "UC" in img.encoding or "FC" in img.encoding:
        log_depth_image(img)
        return

    # decode the image
    if img.encoding == "yuyv":
        log_image(convert_YUYV_to_RGB(img))
    elif img.encoding == "mono8":
        log_image(convert_grayscale_to_RGB(img))
    # continue with all or most encoding types
    else:
        raise_error(f"Unexpected image encoding: {img.encoding}")
```

Listing 4: Pseudocode for Image Callback Function

4.6 Point cloud

Depth cameras can output not only depth images, but also point clouds. These point clouds are a 3d visualization of the depth image, where each pixel is represented as a 3d point in space. Combining this with a colour image, each of these points can also be given a colour, resulting in a coloured point cloud. The code as shown in Listing 5 shows that coloured point clouds and normal point clouds share the same ROS 2 data type, and therefore the same callback function. A coloured point cloud will look like Figure 13 in Rerun.

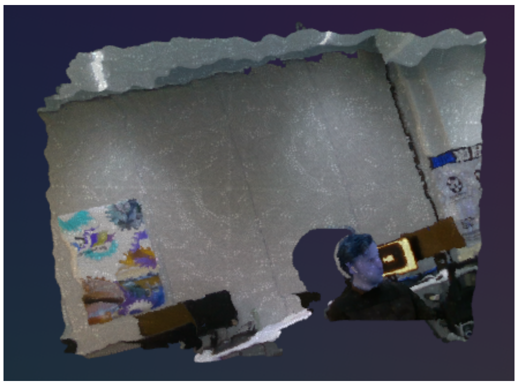


Figure 13: The coloured point cloud

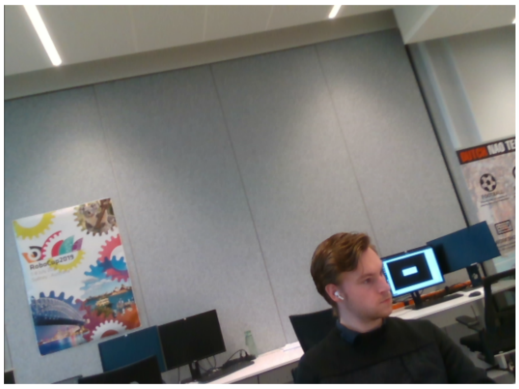


Figure 14: The associated colour image

```
    def point_cloud_callback(points, topic_name):
        pts = point_cloud2.read_points(
            points, field_names=["x", "y", "z"], skip_nans=True
        )

        pts = structured_to_unstructured(pts)

        if "rgb" in [x.name for x in points.fields]:
            points.fields = [
                PointField(name="r", offset=16, datatype=PointField
                    .UINT8, count=1),
                PointField(name="g", offset=17, datatype=PointField
                    .UINT8, count=1),
                PointField(name="b", offset=18, datatype=PointField
                    .UINT8, count=1),
            ]
            colors = point_cloud2.read_points(
                points, field_names=["r", "g", "b"], skip_nans=True
            )
            colors = structured_to_unstructured(colors)
            rr.log(topic_name, rr.Points3D(pts, colors=colors))

        else:
            rr.log(topic_name, rr.Points3D(pts))
```

Listing 5: Pseudocode for Image Callback Function

4.7 Genericity

The best method of proving Swanky is legitimately generic, is to run code that was originally developed for- and only runs on- one robot, and run it on another without changing any of the non-configuration code. With that in mind, the topics shared by the Pupper and turtle-bot, namely the odometry and laserscan topics, were tested on both robots and confirmed that no change in the code was necessary to get both to work. Additionally, the SLAM support in Swanky was originally developed on the Turtle-bot 3. After also trying the node on the Mini Pupper 2 it ran without any problems and was working correctly as shown in Figure 6.

5 Conclusion

Swanky is the first step towards a fully generic Rerun node for visualizing ROS 2 data in Rerun which, although some difficulties were encountered, has proven to be viable in realistic hardware contexts. One of the difficulties is that not every sensor measurement output is easily convertible to Rerun types used for logging. While IMU data can be directly logged to Rerun, for example, custom code had to be written for the LiDAR before it could be logged.

In the future, when wanting to log ROS 2 data to Rerun, using this node will only require adding new data types that are not included yet, but would not require any further setup.

5.1 Future Work

While Swanky is working already, the development in this field is not finished. In this section, we note some possible methods to improve Swanky.

5.1.1 Supporting More Sensors

Swanky currently only supports a select number of sensors types that were available in the Intelligence Robotics Lab (IRL) and that were implementable in the limited time-span. The next step would be to support more default ROS 2 data types. Both more default ROS 2 types and more custom types from additional ROS 2 packages.

5.1.2 Support Custom ROS 2 Types

Currently, Swanky only supports some default ROS 2 data types. ROS 2 does support interfaces, which are used to create custom data types for sensors. Swanky could be extended with support for custom data types. The difficult part, is that it is unknown how the custom data type should be visualized (e.g. in 2D or 3D, or as a LiDAR or a joint angle). How the custom data types are visualized should therefore be configurable in Swanky. This can be done by creating a blueprint that opens when opening Rerun.

5.1.3 Better customizability

With the current setup, the only option is subscribing to a topic, or not. If, for example, some node exports a depth image with the encoding of a normal image, this can not be shown as a depth image in Rerun. For better support, some customizability should be implemented so that problems like these can be handled better by the user, while abstracting away unnecessary or tedious details.

5.1.4 Store data in Rerun

When making a SLAM map, all the data is gathered on the robot on which the map is also constructed. This map is then logged to Rerun, meaning the robot computationally does the heavy lifting. This is not optimal, since a robot often has less RAM and CPU power than the host PC. With the current setup, and only logging the space of the robolab, the cartographer uses 25% of the RAM on the turtle-bot. When trying to map bigger areas, this could become a problem. This problem can be resolved by logging the raw data, and letting the host calculate the map from there. The host (where Rerun runs) can do this using a Rerun plugin, where custom data can be logged. These rerun plugins use the data that comes into rerun, and do extra calculations before it is shown in the Rerun viewer.

5.1.5 Viewer-side ROS message handlers

Regarding rerun plugins, one of the original four ideas for implementing a ROS 2 Rerun interface was implementing a viewer-side message handler in Rerun that gets all the raw ROS 2 data. This would require a ROS 2 node that simply redirects all data to Rerun, which then picks it up and transforms the data in the plugin, so Rerun can correctly show it. The advantage of this would be that the heavy calculations or transformations can be done on the host instead of the client. This would require a whole other project to be set up, but could be a promising new approach.

References

[1] *Improved native ROS support · Issue 1537 · rerun-io/rerun — github.com.* https://github.com/rerun-io/rerun/issues/1537.

[2] Steven Macenski et al. *Robot Operating System 2: Design, architecture, and uses in the wild.* en. May 2022. DOI: 10.1126/scirobotics.abm6074. URL: http://dx.doi.org/10.1126/scirobotics.abm6074.

[3] *Rerun — Visualize multimodal data over time — rerun.io.* https://www.rerun.io/.

[4] *Robot State Publisher.* https://github.com/ros/robot_state_publisher.

[5] *ROS 2 Documentation 2014; ROS 2 Documentation: Foxy documentation — docs.ros.org.* https://docs.ros.org/en/foxy/index.html.

[6] *Use Rerun with ROS 2 — rerun.io.* https://rerun.io/docs/howto/ros2-nav-turtlebot.