Environment Mapping and Reconstruction for Remote Exploration

Jing Wei Nicholas Lim

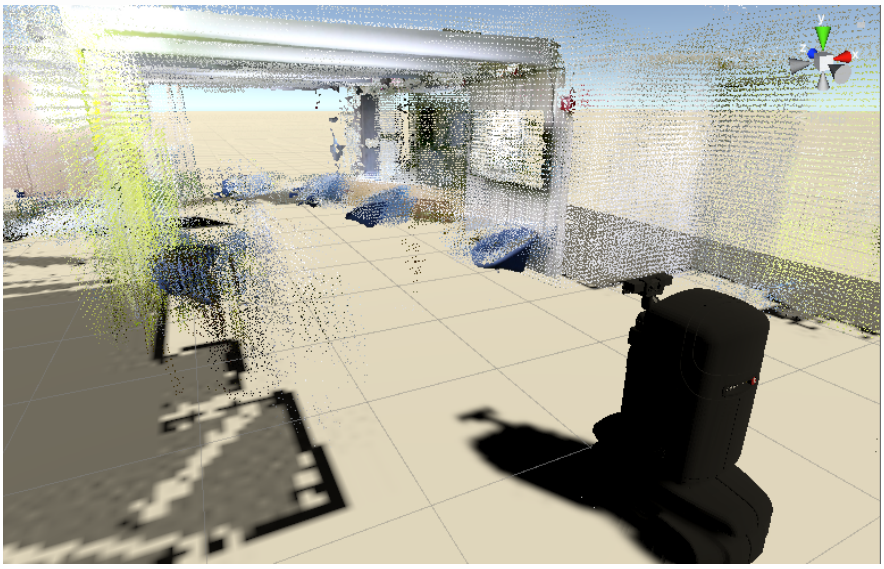

Figure 1: The Movo in the Unity virtual environment that is updated in real-time

Abstract

Remote exploration using a robot opens up the possibility of exploring environments that are unfeasible for humans to be in, such as engaging in rescue operations in disaster zones to space exploration. Current work is usually focused on teleoperation of robots from the point-of-view of the robot - this limits exploration to the physical location of the robot within a remote environment, as well as only visualizing the immediate surroundings of the robot. We propose to use out-of-the-box SLAM to map a remote environment using LiDAR and RGB-D sensor data. We then dynamically reconstruct the map produced by SLAM as a 3D Unity virtual environment that the user can explore using a virtual reality (VR) system like the Oculus Quest and HTC Vive. This mapping and reconstruction is done in real-time while the user is controlling the robot so that the user has live updates of the environment during exploration. Our approach creates a persistent map of the remote environment so that the user can explore the whole environment rather than just the immediate surroundings of the robot. Furthermore, the user is a separate entity from the robot in the Unity environment to allow the user to freely explore the environment without being restricted to the robot's location.

1 Introduction

Robots are powerful tools for exploring remote and hostile environments inaccessible to humans. From conducting rescue operations in disaster zones to surveying unexplored areas in combat situations to space exploration, robots present a safe, viable alternative to direct human engagement. One of the recent advancements in the remote-control of robots is virtual reality (VR) teleoperation. Instead of viewing the robot's camera feed through a flat 2-D screen, VR provides a 6 degree-of-freedom (DoF) view so that the user has a more intuitive experience similar to how humans view the world. We believe that by extending VR teleoperation to remote exploration, we can enhance the user experience in exploration.

One current approach to mapping an unknown environment is using a simultaneous localization and mapping (SLAM) algorithm. However, these maps are often not virtual environments that can be explored. As such, the 3D environment needs to be projected onto a 2D surface like a screen - this is unintuitive for exploration as this is different from how humans view the world. Previous approaches are also mostly focused on VR teleoperation - this means that they only visualize the immediate surroundings of the robot at that point in time instead of generating a persistent map. Furthermore, the user's point-of-view (POV) is the same as that of the robot, hence restricting the user to the robot's physical location within that remote environment.

Our technical approach consisted of three main elements: remote control, mapping and reconstruction. To control the robot (Movo), we first connected the Movo with Unity through ROS-Sharp, which uses Rosbridge as the underlying connector. We then sent inputs from the Oculus Quest controller from Unity to the Movo through ROS-Sharp to move the robot. Next, in order to map the remote environment while the user is controlling the Movo, we used RTAB-Map as an off-the-shelf SLAM algorithm. RTAB-Map uses LiDAR and RGB-D sensor data to produce an occupancy grid map and a global map map.

Lastly, reconstruction of the environment in Unity required 5 streams of data from the Movo: the Movo's transform, the grid map, the color image, the depth image and the global map. Due to the size of these data streams, we faced bandwidth constraints. In the Technical Approach section, we will detail how we overcame these constraints such that the Unity virtual environment is updated close to real-time. We will also cover how these 5 streams of data were rendered in Unity as a cohesive scene for the user.

At the end of this project, we have successfully rendered the 5 streams of data so as to create a persistent map for the user to explore. Through optimizing data transmission, we were also able to provide close to real-time updates to the reconstructed environment in Unity. While we have not run RTAB-Map on the Movo, we were able to run RTAB-Map on a Ubuntu 16.04 virtual machine with ROS-Kinetic, which matches the operating system and ROS version on the Movo. Furthermore, RTAB-Map also successfully mapped level 8 of the SciLi (where the Movo is housed) using a rosbag recording.

2 Related Work

SLAM is already extensively used to map unknown environments. However, these maps are not rendered as virtual environments that can be explored using a VR system. As such, the 3D environment needs to be projected onto a 2D surface like a screen - this is less intuitive for exploration as this is different from how humans view the world. Therefore, this project aims to integrate VR with SLAM by reconstructing the map produced by SLAM as a Unity virtual environment in near real-time so that the user can intuitively explore the environment using a VR system.

In the area of VR teleoperation, there has not been much focus on generating a persistent map in virtual reality. Sumigray and Laidlaw et al.[1] used multi-sensor fusion to improve remote environment visualization for robot teleoperation - however, they did not generate a persistent map of the environment, instead only visualizing the immediate surroundings of the robot at that point in time. We used RTAB-Map as our SLAM algorithm to map the remote environment such that we have a persistent map for the user to explore.

Sekula[2] used a LiDAR point cloud from a Velodyne puck and SLAM to render a remote environment. However, the LiDAR point cloud is only a rough representation of the remote environment with no color such that only the rough outlines can be discerned. We used RGB-D data to perform dense SLAM so that the virtual environment is rendered with more details and hence more realistic. Also, they focused on teleoperation of the Movo - this means that the point of view of the user is always that of the Movo. We aim to separate the user and the robot (Movo) so that the user is free to explore the reconstructed virtual environment.

3 Technical Approach

In this paper, we aim to solve the following problem: given a robot (Movo) in a remote environment, map the environment as the user controls the robot to move within the environment and reconstruct this dynamic map as a Unity virtual environment for the user to explore. To solve this problem, we essentially need to facilitate the following feedback loop (figure 2):

1. The robot collects and sends sensor data to RTAB-Map
2. RTAB-Map generates and transmits map data to Unity
3. Unity renders the map as a virtual environment for the user to see
4. The user sees the virtual environment that is updated in real-time and sends the appropriate movement inputs to the robot (back to step 1)

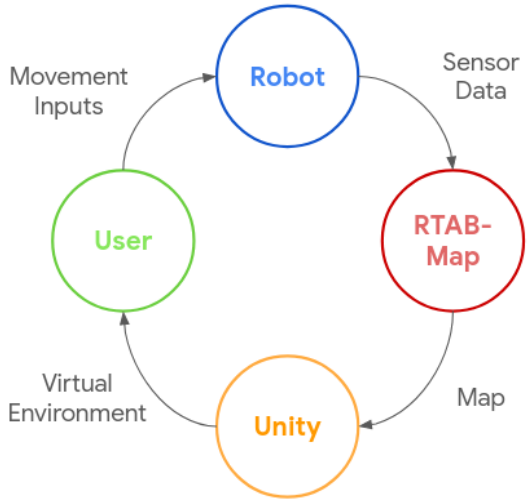

Figure 2: Feedback loop for mapping and reconstructing a remote environment in real-time

In order to facilitate the feedback loop detailed above, we created a software stack outlined in figure 3.

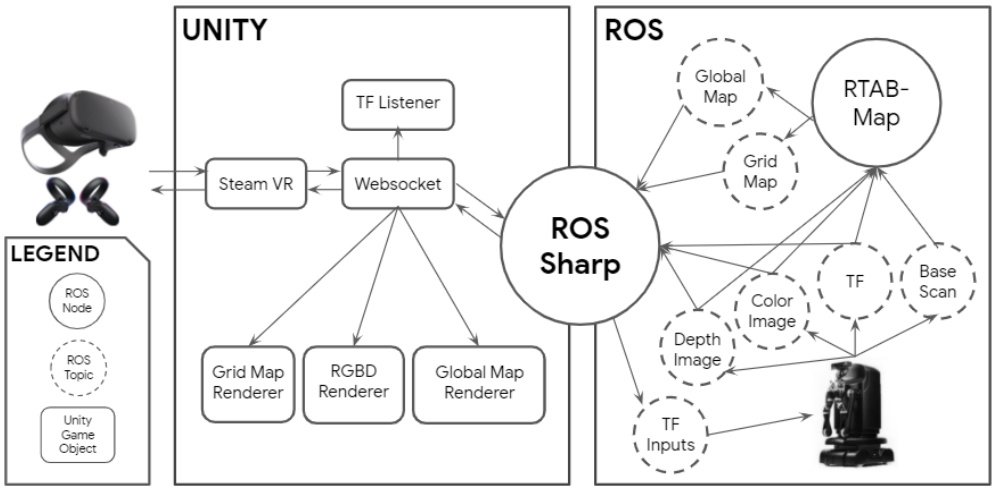

Figure 3: Software stack of ROS, Unity and the VR system (diagram style taken from Whitney et al.[3])

On the ROS side, the robot publishes 4 ROS topics: color image (`sensor_msgs/CompressedImage`) and depth image (`sensor_msgs/CompressedImage`) from the Movo's Kinect2, transform (`tf2_msgs/TFMessage`) and base scan (`sensor_msgs/LaserScan`) from the Movo's LiDAR. RTAB-Map subscribes to these 4 topics and performs SLAM, publishing two ROS topics that constitute the map: grid map (`nav_msgs/OccupancyGrid`) and global map (`rtabmap_ros/mapData`).

We then use ROS-Sharp to communicate between the ROS side and the Unity side - 5 streams of data are sent over to the Unity side: transform, color image, depth image, grid map and global map. The Unity scene has 4 GameObjects that render these 5 data streams:

1. TF Listener: Uses transform to position the virtual Movo within the scene
2. Grid Map Renderer: Renders grid map as a 2D mesh covering the floor
3. RGB-D Renderer: Uses color image and depth image to generate and render a RGB-D point cloud that represents the current view from the Movo's Kinect2
4. Global Map Renderer: Renders map data as a global point cloud representing the map of the remote environment.

As the Unity scene is continuously updated in near real-time, the user views the scene using an Oculus Quest VR system. The user then uses the Oculus Quest controllers to send controller inputs

(`sensor_msgs/Joy`) that are sent over ROS-Sharp and published as movement inputs (`geometry_msgs/Twist`) to the Movo. The Movo moves according to these movement inputs. This completes the feedback loop we outlined in figure 2. In the following subsections, we will go into greater detail on how each part of the software stack was used, as well as the solutions to problems that arose during development.

3.1 Mapping with SLAM

SLAM is essential for mapping a remote and unknown environment as we lack both map information and the location of the robot. We decided to utilize an off-the-shelf SLAM algorithm as they are already mature and robust technologies. We considered a number of SLAM algorithms like RGB-D SLAM, ORB-SLAM2, DVO-SLAM, RGBiD-SLAM and RTAB-Map. We ultimately chose RTAB-Map as it is able to perform dense reconstruction, is being actively developed, has comprehensive documentation and has many useful features.

While mapping, RTAB-Map subscribes to 4 topics published by the Movo: color image (`sensor_msgs/CompressedImage`), depth image (`sensor_msgs/CompressedImage`), transform (`tf2_msgs/TFMessage`) and base scan (`sensor_msgs/LaserScan`). In doing so, RTAB-Map easily achieves sensor fusion of LiDAR and RGB-D sensor data out-of-the-box. Internally, it creates and updates a graph representing the map of the environment. It then publishes two topics: grid map (`nav_msgs/OccupancyGrid`) and global map (`rtabmap_ros/mapData`). These two topics are used as part of the Unity scene rendering that we will explore in the next subsection.

3.2 Rendering in Unity

In order to reconstruct the remote environment as a Unity scene, Unity subscribes to 5 topics that are sent through ROS-Sharp: transform, color image, depth image, grid map and global map. We will elaborate on how each data stream is rendered and the problems encountered.

3.2.1 Transform

The `/tf` transform data stream consists of `tf2_msgs/TFMessage` messages which we use to render the Movo in the correct position in the Unity scene. Each message describes the transform between a parent and child frame. However, when subscribing to the transform topic, we encounter bandwidth issues as there are a lot of frames within the Movo, hence creating a flood of `tf2_msgs/TFMessage` messages. Because we only need the Movo's location within the remote environment, we only require two transforms: `/map->/odom` and `/odom->/base_link`. `/odom->/base_link` provides the location of the robot estimated from the Movo's wheel encoders. Due to inaccuracy from the Movo's wheel encoders, there is odometry drift such that there is some gap between the Movo's odometry and its actual position within the remote environment - `/map->/odom` provides the correction needed. Therefore, to accurately position the Movo within the Unity scene, we add `/map->/odom` to `/odom->/base_link` and set that as the Movo's position and rotation in the Unity scene. To filter out all the other unneeded `tf2_msgs/TFMessage` messages, we wrote a ROS node (`tf_minimal.py`) to filter out messages from `/tf` and only publish `/map->/odom` and `/odom->base_link` messages on the `/tf_minimal` topic. This reduces the number of messages sent over ROS-Sharp and hence avoids the bandwidth issues.

3.2.2 Grid Map

The grid map is a 2D map of the environment taken at the height of the Movo's LiDAR sensor. The following figure shows how the grid map looks like:

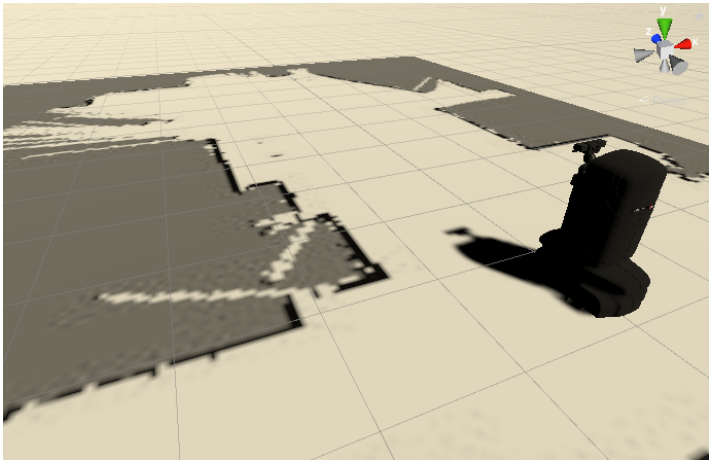

Figure 4: Rendering of the grid map in the Unity scene

We initially transferred `nav_msgs/OccupancyGrid` messages published by RTAB-Map and manually created a texture with the data stored in these messages. However, we encountered bandwidth issues when sending `nav_msgs/OccupancyGrid` messages across ROS-Sharp. This would cause the Unity scene to freeze for a moment every time a grid map is sent. In order to overcome this problem, we created a ROS node (`slam/grid_image_node.cpp`) that first converts the grid map into an image, then compresses it using OpenCV into a JPEG image. As the grid map grows larger, the ratio of compression grows from ~x2 to ~x15. Therefore, compression of the grid map image scales well with the size of the grid map.

Next, we send a `slam/GridImage` message which is a new ROS message we defined to contain `nav_msgs/MapMetaData` (origin of the grid map) and `sensor_msgs/CompressedImage` (the compressed grid map image). As `slam/GridImage` is newly defined, we also need to define it on the Unity side so that ROS-Sharp is able to deserialize the incoming messages.

To render the grid map in the Unity scene, we load the compressed image as a Texture2D, then apply this Texture2D onto a flat mesh with 4 vertices. Lastly, we set the origin of this mesh to be the grid map's origin in the remote environment, which is contained in the `nav_msgs/MapMetaData` field of the `slam/GridImage` message.

3.2.3 RGB-D Image

The RGB-D image is produced by the Movo's Kinect2 as a color (RGB) image and a depth (D) image, providing the current view from the Movo. We chose to display the RGB-D image alongside the global map as it is of a higher resolution and displays updates to the environment with a smaller delay than the global map. The following figure shows how the RGB-D image looks like:

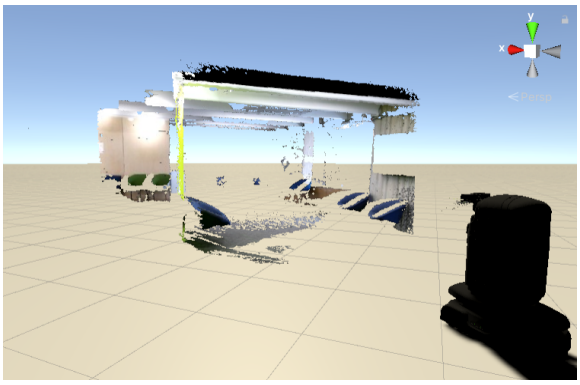

Figure 5: Rendering of the RGB-D image in the Unity scene

To render the RGB-D image, we can choose to either manually integrate the color and depth images in Unity to generate a RGB-D point cloud, or to directly render the RGB-D point cloud generated by RTAB-Map. We originally chose to render the RGB-D point cloud generated by RTAB-Map as it was a simpler task. However, we again faced bandwidth issues when sending `sensor_msgs/PointCloud2` messages across ROS-Sharp due to the sheer size of `sensor_msgs/PointCloud2` messages. This would also cause the Unity scene to freeze every time a point cloud is sent.

To overcome this, we send the color image and depth image as `sensor_msgs/CompressedImage` messages over ROS-Sharp. This avoids the bandwidth issues as the images have been compressed. We then need to manually generate a RGB-D point cloud from the color image and depth image. We first load the color image and depth image as textures - note that we need to take the extra step of decoding the depth image using OpenCVSharp so that we can store it as a 16-bit Red-only texture, whereas decoding the color image is built into Unity.

Figure 6: Color image (left) and depth image (right)

Next, we generate a mesh where each pixel of the color and depth images is represented by a quad consisting of 4 vertices. We use a mesh instead of generating a Unity GameObject for each pixel as too many GameObjects will cause Unity to slow down significantly. In order to apply the color and depth textures to the mesh, we use a shader (reference taken from Laidlaw and Sumigray et al.[1]) instead of setting the depth and color for each vertex manually. This allows us to leverage the GPU for faster rendering as compared to the CPU. The shader takes each vertex of the mesh and scales its position by the corresponding pixel value in the depth texture. It then sets the color of the vertex as the corresponding pixel value in the color texture. In order to form a smooth mesh where nearby vertices are connected, the

shader discards vertices where the difference between its depth and that of the surrounding vertices are too great. However, because the shader is the one that shifts each vertex of the mesh rather than Unity, Unity thinks that the mesh's GameObject is still in its original position. As such, if the mesh's GameObject moves out of the user's camera view despite still being able to view parts of the mesh, the entire mesh will be culled. To overcome this, we need to set the bounds of the mesh's GameObject to be extremely large so that Unity interprets the mesh's GameObject as always being in the user's camera view.

3.2.4 Global Map

The global map is produced by RTAB-Map as a representation of the remote environment. This is the persistent map that that the user can explore, as shown in the following figure:

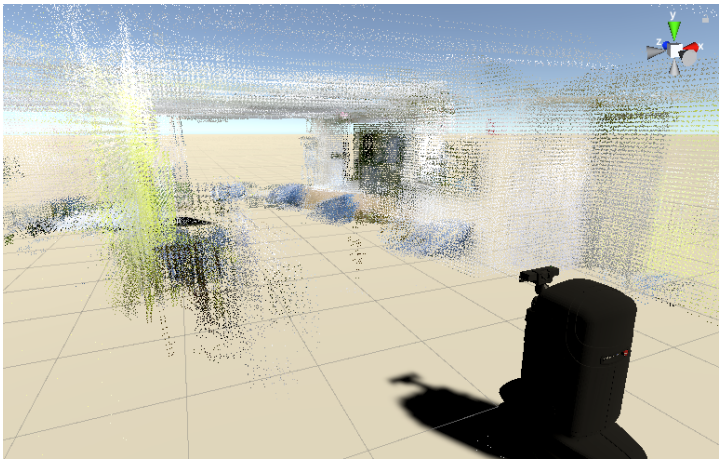

Figure 7: Rendering of the global map (point cloud) in the Unity scene

To render the global map in the Unity scene, we can choose to send either the global point cloud (`sensor_msgs/PointCloud2`) or the global map data (`rtabmap_ros/mapData`). The difference is that global point cloud is a set of points while global map data is a representation of RTAB-Map's internal graph. We chose to send the global point cloud as it is far simpler to parse and render. However, similar to the RGB-D image, we encountered bandwidth issues due to the sheer size of the point cloud. This is especially acute for the global map because it grows in size as more of the remote environment is mapped.

We realized that the `sensor_msgs/PointCloud2` messages published by RTAB-Map contained the whole global point cloud instead of just newly added points. Therefore, most of the points in the

`sensor_msgs/PointCloud2` messages have already been sent in previous messages and hence were redundant. To overcome the bandwidth issue, we then created a ROS node (`pointcloud_update.py`) that converts a `sensor_msgs/PointCloud2` message into a set of points, then calculates the set difference between this new set of points and the previous set of points. This set difference contains all the newly added points, which are then embedded into a `sensor_msgs/PointCloud2` and sent across ROS-Sharp to the Unity scene. This avoids bandwidth issues as the set of newly added points is a small subset of the global map.

However, we realized that there were two issues with directly using the global point cloud: firstly, as the `sensor_msgs/PointCloud2` messages do not contain any information about the direction each point is facing, we are unable to draw a quad for each point since we do not know which direction the quad should face. This results in a rather sparse global map as seen in figure 7. Secondly, when RTAB-Map's internal graph is updated upon loop completion, all of the points in the cloud are shifted. As such, all these shifted points will be interpreted as new points by the ROS node (`pointcloud_update.py`), resulting in the entire cloud being sent. This again introduces bandwidth issues.

To overcome these issues, we decided to render the global map from the global map data (`rtabmap_ros/mapData`). The global map data contains two key pieces of information: the map graph and the RGB-D image associated with each node of the graph. We were able to resolve the first issue as each node has a position and orientation, allowing us to generate a point cloud for each node from its RGB-D image (figure 7) in the same way that we render RGB-D images in section 3.2.3. We were also able to resolve the second issue as we can just shift each node's point cloud that has previously been generated. Sending the global map data does not result in bandwidth issues as each global map data message contains only the latest node's RGB-D image.

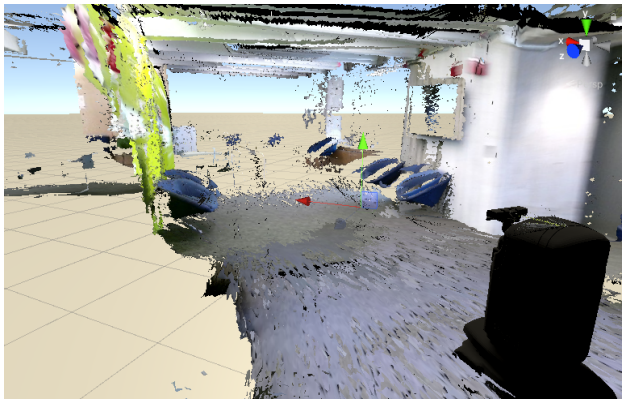

Figure 8: Rendering of the global map (map data) in the Unity scene

3.3 User View and Control

As we want the user to be able to freely explore the Unity scene without being restricted to the Movo's physical location, we separated the user and the Movo as two distinct entities. In the Unity scene, the user and the Movo are instantiated as individual GameObjects. As shown in figure 7, the user is a `Player` GameObject with two hands, whereas the Movo is the `Movo` GameObject. To improve the user experience, we instantiated the `Player`'s hands to be holding Oculus Quest controllers.

Figure 9: User view of the Unity scene

As the user needs to control both the `Player` GameObject and the `Movo` GameObject, two control schemes are needed. We created a custom SteamVR Input binding configuration, where we assigned the left controller for changing the `Player`'s position and rotation, while the right controller sends movement inputs to the actual Movo through ROS-Sharp. As each controller needs to change both position and rotation, we used the trigger button to toggle between changing position and changing rotation. In order to translate inputs from the right controller (`sensor_msgs/Joy`) to movement inputs understood by the Movo (`geometry_msgs/Twist`), we created a ROS node (`movo_teleop_vr.py`) to parse `sensor_msgs/Joy` messages and publish `geometry_msgs/Twist` messages to the `/movo/teleop/cmd_vel` topic.

4 Evaluation

Our goal in building the technical approach above was to enable the exploration of remote environments by using a SLAM algorithm to map the environment and reconstruct it as a Unity virtual environment that can be explored using a VR system. In this section, we will examine the successes of the project so far, as well as its shortcomings.

4.1 Successes

In regards to mapping, we successfully mapped the 8th floor of the SciLi using RTAB-Map as our SLAM algorithm. Mapping was done close to real-time in a Ubuntu 16.04 virtual machine with ROS Kinetic and playing a rosbag that recorded a session of driving the Movo around on the 8th floor of the SciLi. This shows that the Movo publishes all the topics required for RTAB-Map to map a remote environment using LiDAR and RGB-D sensor data.

In terms of rendering, we were able to transfer all 5 streams of data (color image, depth image, transform, grid map and global map) from ROS to Unity in close to real-time. We were able to achieve this by overcoming significant bandwidth constraints through optimizing the transmission of data from ROS (in the Ubuntu 16.04 VM) to Unity through ROS-Sharp. We were also able to render these streams of data in the Unity scene to form a cohesive virtual environment for the user to explore. We believe that this is an improvement on current approaches for mapping and reconstructing a virtual environment (figure 10).

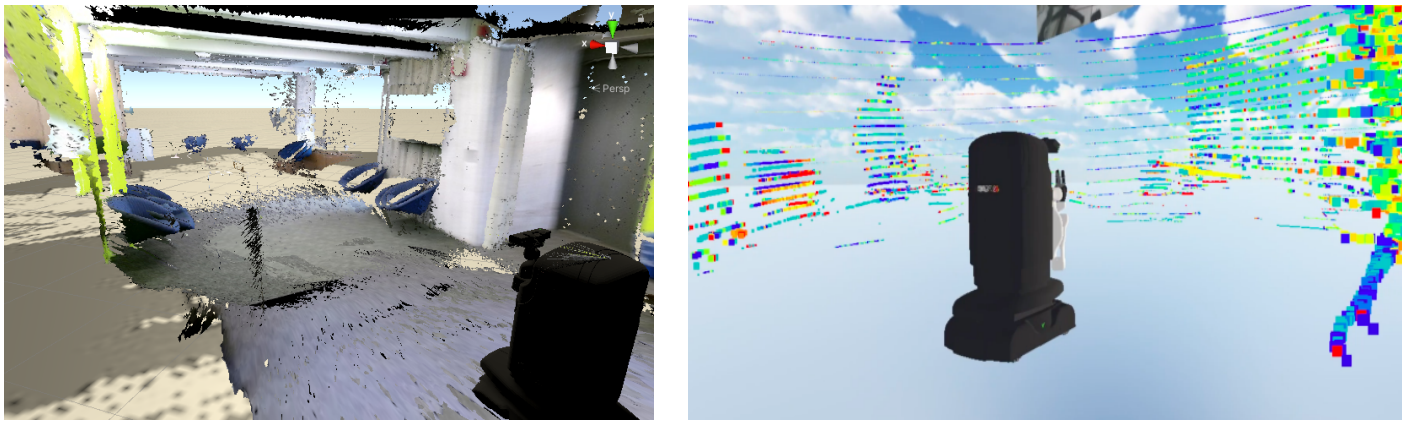

Figure 10: Comparison of map reconstruction between our project (left) and Sekula[2] (right)

Lastly, with respect to user view and control, we created a `Player` GameObject that was able to control both the `Movo` GameObject and itself. We also verified that the right controller was able to send movement inputs that were correctly translated by our ROS node (`movo_teleop_vr.py`) by moving the Movo in a Gazebo simulation. The changes in position and rotation of the Movo in the Gazebo simulation were also accurately reflected in the Unity scene.

4.2 Shortcomings and Potential Improvements

One shortcoming is that we have not been able to run the project on the actual Movo. For some yet-unknown reason, RTAB-Map is not receiving any messages published by the Movo, despite us being able to echo the messages on the command line. As such, we have only ran the project on a Ubuntu 16.04 VM and ROS Kinetic with a rosbag recording. We foresee that when we move from the rosbag recording to the actual Movo, one potential issue might be further bandwidth constraints. As of now, because Unity and the ROS VM are running as separate applications on the same laptop, the network connection is stable. If we run the project on the actual Movo, Unity and ROS will be running on separate machines where Unity is on the laptop while ROS is on the Movo. This might introduce further bandwidth issues as the network connection between different machines is less stable.

Another thing we can improve is the rendering of the global map. As seen in figure 11, some of the nodes are not positioned correctly, resulting in an inconsistent scene. We believe that RTAB-Map filters and removes nodes from global map data before rendering to produce a more coherent scene (figure 11 right). Therefore, one way to improve our rendering of the global map is to filter and remove outdated nodes.

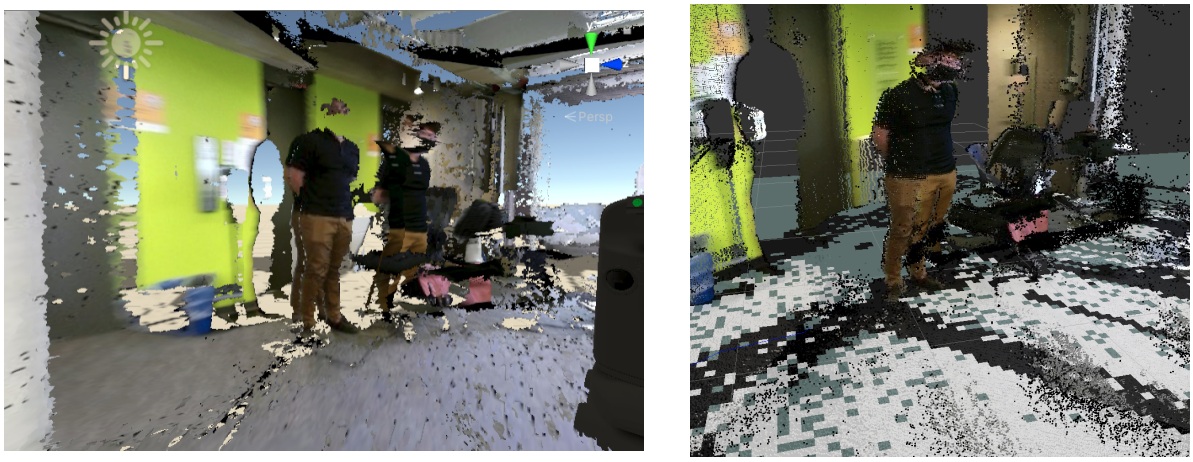

Figure 11: Problems with rendering the global map data (left) vs. rendering by RTAB-Map in RViz (right)

5 Conclusion

Through this project, we are now able to map a remote environment and reconstruct it as a Unity virtual environment for the user to interact with. Our Unity scene is able to receive 5 streams of data from ROS and render them close to real-time despite facing bandwidth constraints by optimizing data transmission. However, as noted in the Evaluation section, there are a few shortcomings that provide a starting point for improvements.

Beyond these shortcomings, there are various paths for future work. One such path is improving the transmission of data from ROS to Unity. One problem we consistently faced throughout this project is the bandwidth constraints when transmitting each data stream from ROS to Unity. To overcome these constraints, we need to compress some of the data, resulting in the loss of information. A possible solution is to parallelize the transmission of data, or to find a better alternative to ROS-Sharp's connector. This will allow us to transmit higher quality data, such as higher resolution grid images and RGB-D images. This will also enable us to operate in situations with even greater bandwidth constraints, such as exploring the ocean floor or working in disaster zones.

One other possible path is the automation of exploration and pathfinding. In situations where we are unable to overcome bandwidth constraints such that real-time mapping and control of the robot is impractical, we can allow the robot to autonomously explore the remote environment. After it has explored and mapped the environment, the map can be transmitted back to Unity for rendering. We can then use this map for pathfinding so that the robot can move to selected locations in the map without real-time control by humans.

Another potential path is integration with VR teleoperation. While mapping itself has its uses, integration with VR teleoperation will allow the user to not only explore the remote environment, but also to manipulate objects in the environment. Furthermore, the manipulation of objects can also enhance exploration by opening up new areas for the robot to explore, such as opening doors or removing obstacles.

Lastly, we hope that mapping and reconstruction of remote environments using robots will become sophisticated enough such that it becomes a viable alternative to human exploration in hostile and dangerous environments.

6 Bibliography

[1] Sumigray, A., Laidlaw, E., Tompkin, J., & Tellex, S. (2021). Improving Remote Environment Visualization through 360 6DOF Multi-sensor Fusion for VR Telerobotics. *Companion of the 2021 ACM/IEEE International Conference on Human-Robot Interaction*. doi:10.1145/3434074.3447198

[2] Sekula, A. (2019). Improving Movo Teleoperation in VR with Lidar.

[3] Whitney, D., Rosen, E., Ullman, D., Phillips, E., & Tellex, S. (2018). Ros Reality: A Virtual Reality Framework using Consumer-Grade Hardware for ROS-Enabled robots. *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. doi:10.1109/iros.2018.8593513