Improving Movo Teleoperation in VR with Lidar

Alex Sekula — May 16, 2019

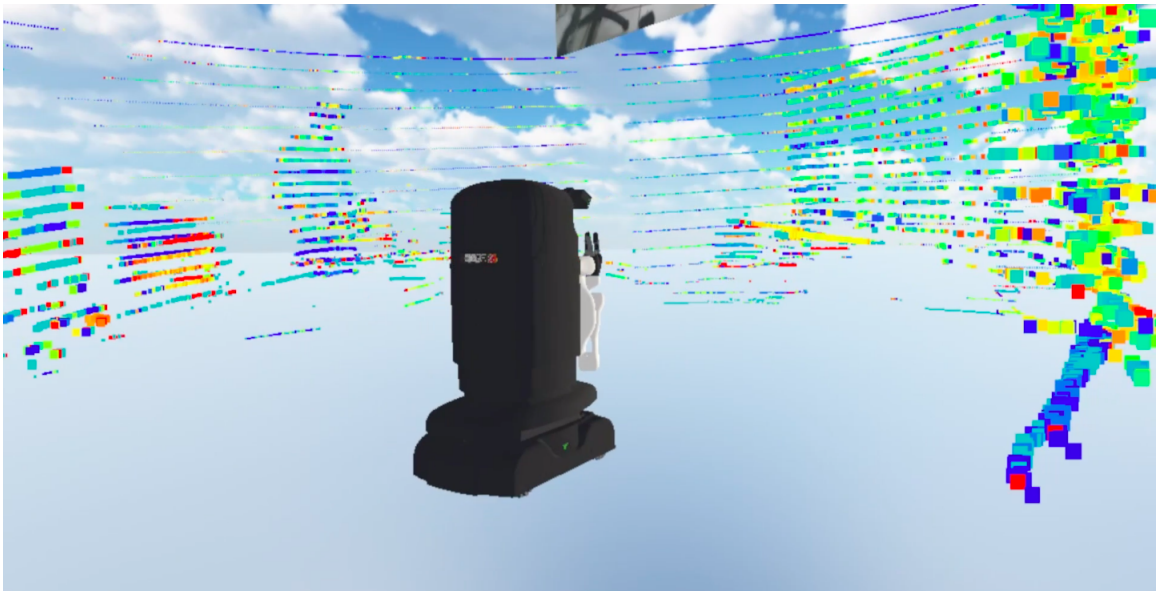

Figure 1: The Movo in VR surrounded by the rendered Lidar point cloud.

Abstract

Robot teleoperation using VR has the opportunity to improve user experience in controlling robots. Current work has developed VR teleoperation that renders an environment that is not fixed in VR space, which has the potential to cause nausea and confusion in users. In this project, we improve the user experience of VR teleoperation using Lidar and SLAM. We use Lidar to provide a more complete view of the environment, which allows users to plan actions better in dynamic environments. Furthermore, we use out-of-the-box SLAM to transform the Lidar points in VR as the robot moves, giving the effect that the virtual robot is moving within a fixed environment in VR. This work provides a user experience where the user feels like they are in the robot's real environment, as opposed to prior work where the user felt like they were viewing the environment projected in front of a fixed robot. Our new approach has the potential to improve success of certain tasks that involve navigating the Movo in a dynamic environment, and interacting with the environment (e.g. opening doors).

1 Introduction

VR teleoperation is a new form of teleoperation of robots. The use of VR allows for a richer display of the robot's state and environment, and enables the users to control the

robot in a more intuitive way. Many robots today are still operated by humans, and a better user experience can improve the success rate and time spent on certain tasks. Furthermore, teleoperation provides a framework to collect data that may be used to train robots to autonomously perform tasks in a supervised manner.

There have been two main approaches to this task. The first approach was not performed on a mobile robot, and thus did not explore the extra challenges and opportunities that arise with teleoperating on a mobile robot. The second approach used sensor data that did not provide a complete environment when viewed in VR, and did not use the robot's location data to adapt the sensor data to the robot's location.

Our technical approach consisted of two main elements: Lidar and SLAM. The first part of the project consisted of getting Lidar data from the Movo rendered in the VR environment. This approach involved connecting the Lidar to the Movo, connecting the Movo to Unity through Rosbridge, calibrating the points, and then rendering the points in Unity. While most of this part of the project involved connecting many technologies together, many problems arose in the process and in this paper we outline solutions to those problems.

The second main element of the process was using the Movo's location with the Lidar point cloud, so that when the Movo moves the environment as rendered by the point cloud does not move with it. In other words, suppose a chair was sitting right in front of Movo, and the chair was being renderd in VR. Were the Movo to move closer to the chair, we aim that in VR space, the chair would not move, but the Movo would move closer to the chair, effectively mimicking reality. This approach consists of writing a manual publisher on ROS to publish a transform along with a timestamp of the transform, and then synchronizing the Movo's location and point cloud on the Unity side, so that the point cloud changes at the same time as the Movo moves. We describe the importance of synchronizing the data streams, and we explain the technical details of the synchronization process.

We lastly note a number of successes of the project, including a better interface for viewing the Lidar point cloud, and a successful synchronization method for Lidar and SLAM data within Unity. The resulting project achieved the goal of rendering an environment that adapts to Movo's movements, providing users with the effect that they're in a virtual copy of the Movo's environment. We also detail some shortcomings and future improvements of the project, noting in particular the lag between when the Movo senses changes in the environment to when those changes are rendered in VR, as well as slight jitter in the point cloud due to an imperfect synchronization method.

2 Related Work

There have been two main approaches to VR teleoperation. Whitney et al.[1] applied VR teleoperation to Baxter to pick up objects. Their approach did not generate an entire

environment around Baxter as they were only concerned with the objects immediately in front of it, nor did they teleoperate with a mobile robot. In doing so they failed to examine a key advantage of VR teleoperation, namely moving mobile robots in a dynamic environment. The second main approach, done by Nishanth Kumar, applied VR teleoperation to the Movo, and generated the Kinect RGBD feed as a point cloud in front of the robot.[2] The shortcoming in that approach was that the Movo was fixed in VR space, and whenever the Movo moved in real life, the point cloud would move in VR space instead of the robot. This provides an unnatural user experience: a main benefit of VR is to put the user in a simulated environment that feels and acts like a real environment. An environment that rotates and translates about the user while the user position remains fixed, as implemented Kumar's work, does not take advantage of a key strength of VR.

3 Technical Approach

The problem we aim to solve is as follows: given a Movo in an environment, with a 360 degree Lidar sensor situated on the robot, we aim to render the robot and Lidar data in virtual reality and control the robot via virtual reality controls. Furthermore, when the Movo moves, we aim to move the virtual Movo in VR, and render the point cloud such that the location of static objects remain fixed in VR space. In other words, while the point cloud may change, fixed objects that the point cloud renders in VR should not move in VR when the robot moves.

We provide a diagram outlining the software stack that we used to connect VR and the Movo, so that we could control the Movo from VR, and view the Movo's sensors in VR. A diagram can be found below.

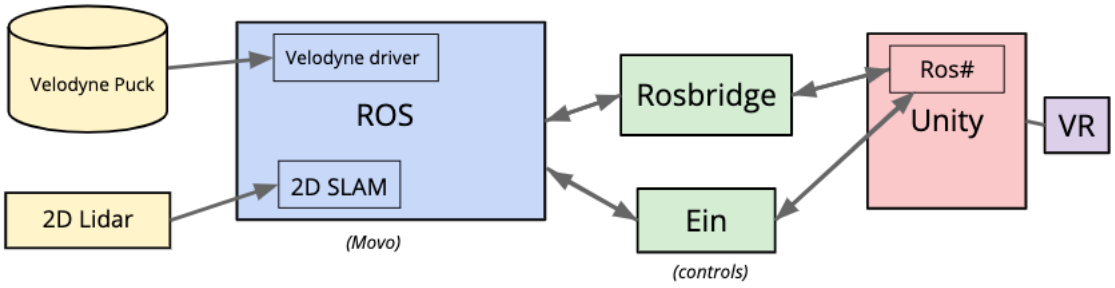

Figure 2: The software stack used to connect VR with the Movo.

In this section, we go into detail on how we used each part of the software stack, as well as solutions to problems that arose during the development process. Our approach consists of two main subsections: the first section details how we render the Lidar point cloud in VR, and the second details how we incorporate SLAM in VR to update the VR environment according to the Movo's movements.

3.1 Lidar to VR

We make use of a more complete representation of the environment by using a Velodyne VLP16 Puck[3] that uses LIDAR to scan the environment. Using the puck, we get access to a real time point cloud representing the environment in 360 degrees around the Movo. This provides a better representation in VR than using the Kinect which only provides environment data in front of the robot. A more complete representation allows users to navigate better, especially in the presence of dynamic objects.

3.1.1 Initiating the Velodyne/Movo Handshake

The first main challenge arose from connecting the Velodyne Puck to ROS, so that they can communicate. The Velodyne by default is designed to connect to a system with local IP addresses in the `192.168.x.x` range, whereas ROS operates within the `10.66.171.x` range. Upon connecting the Velodyne to ROS, they were unable to ping each other. After examining packets via Wireshark, we found that sending a ping broadcast from the Movo to address `192.168.1.255` initiated an ARP request and response between the Velodyne and the Movo, which enabled direct pings between the Movo and the Velodyne. This also enabled use of the Velodyne's user interface, which allowed us to change the Velodyne's default IP address to one within the `10.66.171.x` range. This allows the Velodyne to automatically connect to the Movo when the Movo turns on, without requiring manual ping broadcasts.

3.1.2 Accessing the Point Cloud in Unity

After the Velodyne connected successfully with the Movo, the next step was to get the Velodyne point cloud data into Unity. On the ROS side, this step was mostly seamless: we first installed the drivers from the Velodyne site, and then ran the launch script, which published the point cloud as a `sensor_msgs/PointCloud2` type on the rostopic `/velodyne_points`. On a machine on the network in the lab, we ran rosbridge[4] which ran a websocket converting the rostopic data on the Movo to JSON objects that can be accessed by anyone on the network.

On the Unity side, we made use of ROS#[5] and the built-in `PointCloud2` data type to access the websocket data within Unity. We wrote a class that read messages from the websocket, and stored them in a `PointCloud2` object. This allowed us to have continual access to the Lidar data within Unity. Upon each new point cloud that the Velodyne read, the point cloud would be published on `/velodyne_points` and registered by the Rosbridge websocket, then sent from the websocket to the Message Receiver class within Unity.

We note one small issue that arose with the PointCloud data: while the `sensor_msgs/PointCloud2` takes in rgb data, the Velodyne broadcasts the rgb data as a scalar between 0 and 255

as the blue value, and zeros for the red and green values. This is because the Velodyne does not capture color, yet still provides a value for each point to aid in visualizing the point cloud. We applied the same method as rviz in coloring the points: we converted the value to a color along a spectrum ranging from blue to red, passing through green and yellow respectively.

3.1.3 Rendering the Point Cloud in Unity

Once the point cloud data was accessible in Unity, the next step in this section of the project was to render the point cloud within the VR environment. We aimed to avoid creating a GameObject for each point in the point cloud, as the number of points was too large (around 20,000 points per point cloud), and generating 20,000 GameObjects would significantly slow Unity down. Instead, we created a Mesh and MeshRenderer in the Unity scene, and used a custom shader designed specifically for point clouds to render the point cloud. The shader was taken from a Unity internet forum.[6] The only modification made to the shader was to change the rendered points from triangles to squares. Once the shader was incorporated into Unity, the Lidar point cloud was rendered in Unity at each timestep.

3.1.4 Calibration

The last step in this section was to calibrate the point cloud so that the points in VR were aligned correctly with the user's perspective. In other words, we aimed to render the environment as if the user were standing in the real environment, with the virtual floor as rendered by Lidar points beneath the user, the virtual ceiling above the user, and the entire virtual environment level with the user's eyesight. We note that this did not occur automatically — as the focus of this project was not on calibration, we refrained from incorporating automatic calibration techniques and manually transformed the points to their correct position in VR.

The first calibration step required publishing a static transform between the Movo's `base_link`, representing the position of its base, with the Velodyne puck. To do this, we ran `rviz` and enabled the Kinect point cloud along with the velodyne point cloud. We published a static transform and tweaked the transform values until the velodyne points matched up with the Kinect points. An image of this effect can be found below, as well as a link to a video of this process can be found at https://youtu.be/HJ6vcdtEmo8.

Once the Velodyne points were successfully calibrated on ROS, the points needed to be calibrated within Unity as well. Note that two rounds of calibration are not needed: we could have arrived at the same final calibration of points were we to skip calibration on ROS, yet for the sake of future projects that may use the Velodyne points, we calibrated on ROS as well. We also note that once the points were calibrated on ROS, they were not automatically calibrated in Unity as well: in Unity, the y- and z-axes are swapped

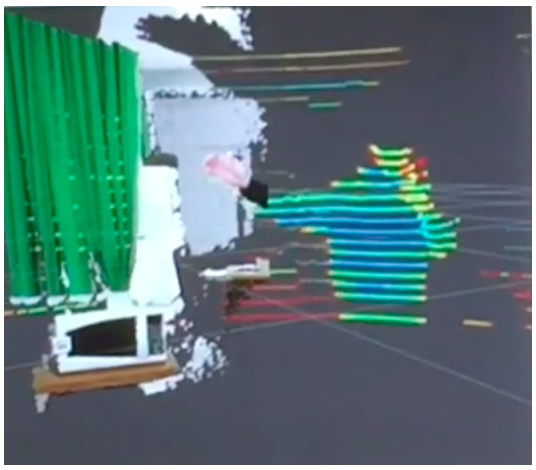

Figure 3: Lidar and Kinect point clouds overlayed for the purposes of calibration.

from ROS's coordinates (i.e. "up" in ROS is the z-coordinate, whereas "up" in Unity is the y-coordinate), and even merely swapping the y and z values of the point cloud did not yield the correct positions. We repeated the process of calibrating the points in Unity as we did in ROS. First we enabled the Kinect point cloud in Unity, then we manually transformed the Velodyne points until they lined up with the Kinect. We are unaware of why we needed to perform such a calibration, and future investigation into the problem may yield an answer.

Once the points were calibrated, and upon running rosbridge, the velodyne launch script, and the Unity scene, the point cloud from the Velodyne puck was successfully rendering in quasi-real-time in the VR environment, and the rendered environment resembled the Movo's environment. Note that the points were not quite rendered in real time due to lag in sending the Velodyne data from ROS to Unity. We describe the lag in more detail in the Evaluation section.

3.2 Incorporating SLAM

The second main part of the project was to incorporate the Movo's real-time location within the environment, and transform the Movo and point cloud in VR according to the updated position. In other words, we aimed to move the virtual Movo in VR exactly how it moves in real life. In this section, we first describe motivation as for why this creates a much better user experience, then we explain the technical details of how we incorporated SLAM.

3.2.1 Motivation

The main reason why incorporating SLAM into VR teleoperation of a mobile robot is useful is that in doing so we build a fixed environment in virtual reality that represents the robot's environment. Were SLAM not incorporated, if the robot were to move around, then it would appear in VR that the environment is rotating and translating about the robot, and that the robot was fixed in space. We show this visually below.

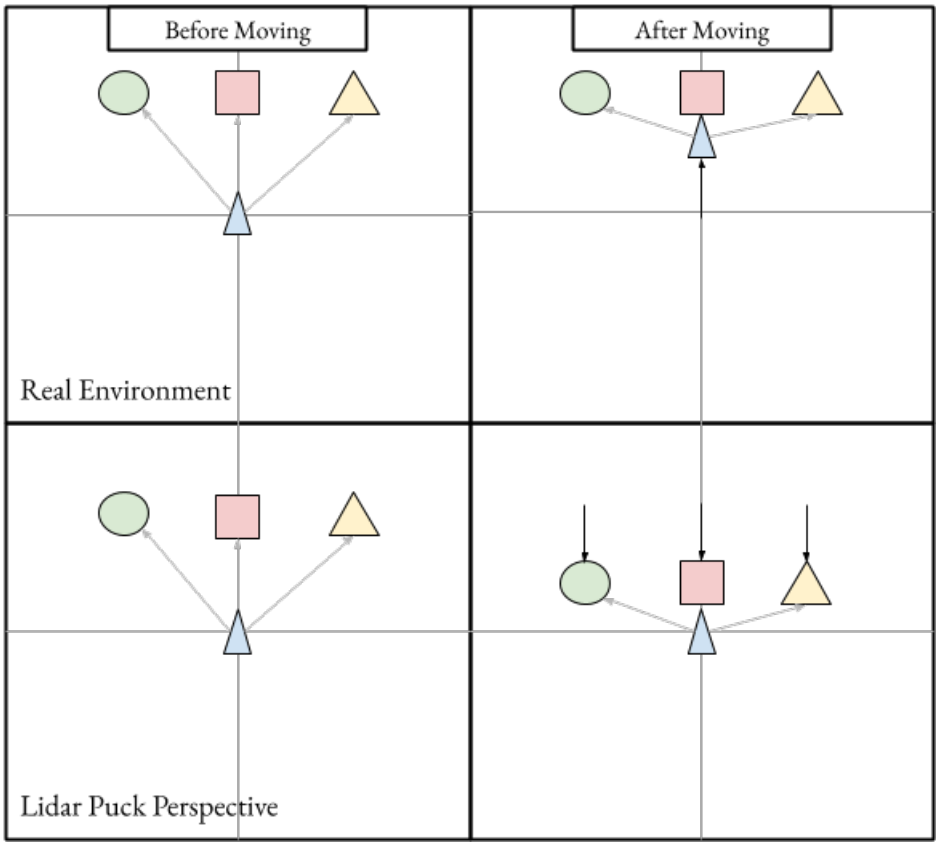

Figure 4: Before and after the Movo moves, as seen in the real environment versus from the Lidar puck's perspective.

When the robot moves forward, the Lidar point cloud observes that the objects move backwards. Therefore, if the point cloud was displayed in a fixed location in VR, the points would give the visual effect of a shifting environment whenever the robot moves. To counteract this effect, we would need to move the point cloud in VR wherever the Movo moves.

If the robot is in position $\vec{p_1}$, and an object is in position $\vec{x}$, then the Lidar will register the object relative to the Movo at position $\vec{x} - \vec{p_1}$. If the robot moves to position $\vec{p_2}$, then the point cloud will register the object at position $\vec{x} - \vec{p_2}$. If we fix the Lidar point cloud in VR space, then the object will move from $\vec{x} - \vec{p_1}$ to $\vec{x} - \vec{p_2}$, effectively

translating $\vec{p_1} - \vec{p_2}$. By shifting the point cloud when the robot moves by $\vec{p_2} - \vec{p_1}$ in VR, then the object will appear to have moved $(\vec{p_1} - \vec{p_2}) + (\vec{p_2} - \vec{p_1}) = \vec{0}$ in VR. In other words, the object will not appear to have moved.

The same can be said for rotations: if the Movo rotates θ, then the point cloud needs to also be rotated θ in order to counteract the effects of the moving Lidar puck. In other words, in VR the point cloud needs to move exactly how the robot moves in reality.

As shown, the robot's position is necessary in rendering objects consistently in the environment. In the next sections we detail how we obtained the robot's location, and how we rendered the point cloud according to the location data.

We briefly note that the Lidar point cloud does not completely capture an object, and that if the Lidar puck moves closer to an object the Lidar may capture points from different parts of the object. In practice we observed that the closer the Lidar puck gets to an object, the closer the Lidar rings get, and so while a fixed object may be rendered at the same position in VR space when the robot moves, the points that render the object may move on the object itself. This effect can be observed in the second demo video in the Evaluation section. In the last section of this paper we detail future work that may alleviate this effect.

3.2.2 SLAM Messages

To get the location data from the Movo into Unity, we first ran out-of-the-box SLAM using the 2D Lidar data from Movo's base. The launch script published a transform between `/base_link` and `/map`, which described the Movo's position in the map. Since ROS# does not have access to Movo's transform tree, we wrote a manual publisher that read the transform and published it as a `geometry_msgs/transform_stamped` message on a rostopic which included a timestamp of the transform. This allowed the location data to be passed through rosbridge into Unity. We lastly wrote a ROS# Message type `GeometryTransformStamped` so that ROS# could automatically parse incoming messages from the rosbridge websocket.

Once the location data was received in Unity, we transformed the Movo's position according to the position within the map. We set the point cloud object as a child of the Movo within Unity, so that whenever the Movo rotates or translates in VR, the point cloud moves with it to provide the effect of rendering a fixed environment.

3.2.3 Synchronizing Data Streams

We note one crucial aspect of incorporating location data with the Lidar point cloud: in order to maintain the effect that the Lidar points render fixed objects in the same location, we must transform the Movo in VR and render the Lidar point cloud using sensor data taken *at the same time step*. Suppose the input streams were not synchro-

nized, and the location data was received in Unity one second before the Lidar data was. Then suppose the Movo were to move forward: the Movo would be shifted in VR, and therefore so would the point cloud, yet the updated point cloud would not have been received so it would appear as if the entire environment shifted away from the Movo. One second later, the updated point cloud would be received in Unity and then the environment would shift back. In other words, the environment would shift forward, and then backward one second later to its original position. This effect would be quite confusing to the user, and it nullifies the entire point of incorporating SLAM.

To prevent this issue, we examined the timestamps of the incoming messages in order to use the Lidar data and location data at roughly the same time. Since the location data was a lot smaller, the messages were coming in faster, and therefore we stored the location data on a FIFO queue. Once a point cloud was received in Unity, we popped positions off of the queue until the timestamp of the location message at the head of the queue was later in time than the incoming point cloud. We used the last popped location to set the Movo's location, and we updated the point cloud immediately with the new point cloud data. This method was accurate within around 0.1s of data, i.e. whenever a point cloud was rendered, the Movo's position data was usually at most around 0.1s behind the point cloud data. We detail future improvements to this method in the Evaluation section. We note that since the data was never perfectly matched up, we observed slight jitter in the environment as rendered by the point cloud whenever the Movo moved.

4 Evaluation

Recall that the goal of the project was to build a VR demo that renders the Movo's Lidar data as a point cloud, and uses SLAM to transform the virtual Movo and point cloud in VR according to how the Movo moves. In this section, we first examine successes that we attained in the final demo, and then describe shortcomings of and potential improvements of the project.

4.1 Successes

We list three main successes:

1. **Lidar Point Cloud Rendering**: The final project successfully rendered the Lidar point cloud in VR, and the point cloud as viewed in VR resembled the Movo's environment. We note in particular the improvement upon viewing point clouds on a monitor. By utilizing the unique way that VR handles depth with two monitors, one for each eye, the final demo allowed users to more easily distinguish closer points from farther points, as opposed to a monitor where depth is harder to recognize. We

also note that the virtual reality user experience gives the user the feeling that they are inside the environment, which provides the user with a better intuitive understanding of the environment. The VR head controls also provide more intuitive control over viewing the environment than using a mouse to rotate and translate the users' perspective. Lastly, the use of Lidar provides a more complete rendering of the environment than the Kinect point cloud, and a more complete rendering has the potential to make navigation tasks easier.

2. **Moving the Virtual Movo in VR**: The final demo successfully moves the virtual Movo in a way that resembles how the Movo moves in reality. In the shortcomings section, we will describe a small transform bug that was not fixed in the final demo, yet barring the bug the virtual Movo's movements match the real Movo's movements in the environment, and the point cloud location moves correctly along with it, so that the environment appears fixed in VR space. This was a major success, as this was the main goal of the second half of the project.

3. **Synchronizing the Lidar and SLAM Data**: While the final demo contains a bit of jitter in rendering the point cloud as representing a fixed environment in VR, the Lidar and SLAM data streams are synchronized, and the resulting point cloud renders objects as if they are fixed in VR space even when the Movo moved in reality. This was also a success because without this the demo would not give the user a sense of a fixed environment, which would worsen overall user experience. Furthermore, we anticipate inducing nausea in some users if they were to view a virtual environment that translates and rotates frequently due to unsynchronized data streams.

4.2 Shortcomings and Potential Improvements

We list a few shortcomings of the project, as well as potential solutions for future work:

1. **Lag**: We note that a major drawback of the final demo is the lag between the real time environment and the point cloud in VR. There is around five seconds of delay between when an event happens in the Movo's environment to when the event is rendered in VR. Furthermore, the lag increases as the demo progresses. For the purposes of producing a proof-of-concept for the final demo, other features of the project were prioritized, yet we propose that future work on this project begin with reducing lag. A few approaches that may be effective are a) to profile the Unity code for inefficiencies, b) to compress the Lidar point cloud in ROS so that less data is sent over the network, and c) to reduce the frame rate of the point cloud. Lag reduces usability of the project, and while it is acceptable in a demo, it needs to be drastically reduced before the project could be considered a finished product.

2. **Environment Jitter**: As mentioned earlier, the imperfect data synchronization method yields slight jitter in the environment as the Movo moves. A simple fix

would consist of writing a publisher on ROS that reads the location data and sensor data at the same time step, and publishes them together as a single message so that no synchronization is required on the Unity end. We predict that such a fix would eliminate jitter altogether.

3. **Location Transform Bug**: We note that in the final demo there's still a transform bug that causes the virtual Movo to mirror the real Movo's movements about the y-z plane in Unity. In other words, when the real Movo moves left the virtual Movo moves right, and vice versa. Note that the point cloud also has this bug, which is how the demo can render the environment as fixed when the Movo moves despite the bug in the virtual Movo's movements. We anticipate that more time spent debugging transforms would eventually fix this issue.

4. **A Brittle Software Stack**: We lastly note a major drawback to general development on the software stack used in this project. Namely, the entire stack is brittle and prone to breaking. In some cases, the Movo failed to run its launch script correctly, and required a few reboots before it ran successfully. In other instances, rosbridge failed to detect rostopics on the Movo. Another issue consisted of out-of-order packets from rosbridge in Unity. Further issues involved problems with the VR controllers, as well as the headset failing to display the VR environment. All of these kinds of issues extend development time, and we raise the question of how much time should be spent in developing projects atop the software stack versus developing less error-prone software. We do not attempt to answer such a question. We do, however, warn future developers of these issues so that they may consider whether their projects are feasible given their timeframe and the general delay caused by many unforeseen breaks in the software stack.

4.3 Demo Videos

We include two demo videos. The first shows the Lidar point cloud rendered in VR, alongside a video of the environment around the Movo. The second shows how the point cloud adapts to changes in the Movo's position, demonstrating that the rendered environment does indeed appear to be fixed regardless of the Movo's movements. Note that in both videos, the overlayed videos have been edited to match events in VR with events in reality, whereas in the actual demo the events occurred with seconds of lag.

The first video can be found at: https://youtu.be/UAxVMhP1Gw0, and the second can be found at https://youtu.be/XWc4mgR5bMg.

We include a few screenshots of the demo videos below.

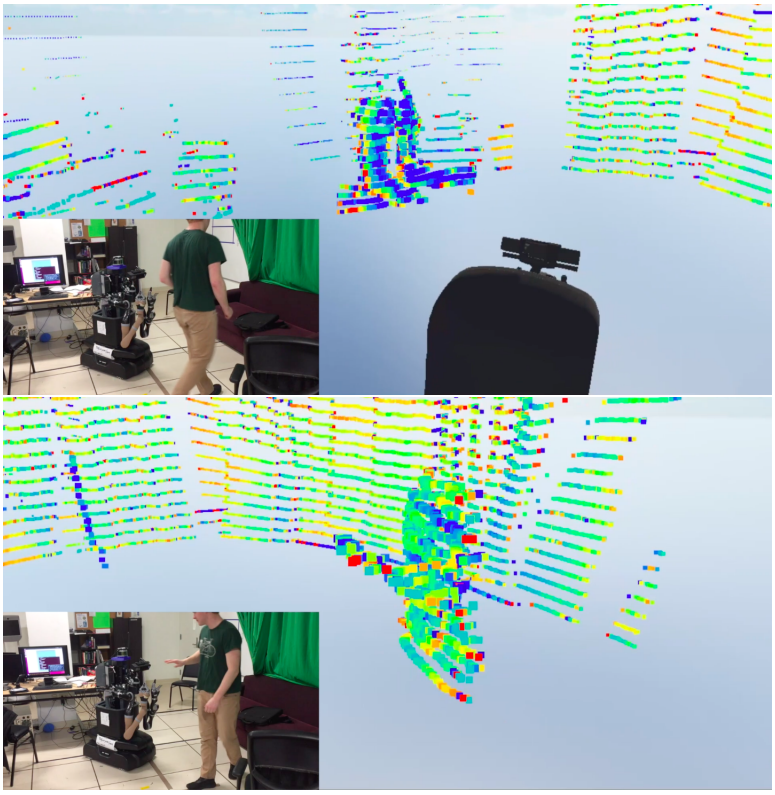

Figure 5: Screenshots of the final demo.

5 Conclusion

We have detailed the final demo, which successfully renders the Movo's Lidar point cloud in VR, and renders the point cloud and virtual Movo according to the Movo's movements. We have made use of a pre-existing software stack, and have extended many aspects of it to fit the use case in this project. We have addressed problems that arose in the development process, including networking issues as well as data stream synchronization. The project successfully achieved the goals set out at the start of the project, but still contains many drawbacks that provide a starting point for future work.

After the small issues detailed in the Evaluation section have been addressed, we envision a number of additions to the project that we would like to see. The first addition is to perform basic texture mapping atop the point cloud. While users can discern the environment from the point cloud alone in VR, the overall look and feel of the point cloud would improve if textures are rendered atop the points. Instead of people being rendered as a collection of points, they could be rendered as humanoid meshes. This has the potential to make the VR-rendered environment look more like the Movo's real environment. This would require an efficient method for performing texture mapping

on a sparse point cloud.

Another addition that would provide useful is to perform real-time 3D mapping using the point clouds over many timesteps. We envision that we could write a back-end server that can store and process the point cloud, and implement 3D mapping techniques using the Movo's 2D SLAM data. We also hope that such a server could store large maps so that the Movo could map entire floors of buildings and store the maps so that users could revisit the maps in VR offline or the next time that they teleoperate the Movo.

Lastly, we hope that in the long term, user interfaces for teleoperation user interfaces will improve enough that users can easily teleoperate robots for performing daily tasks, and that teleoperation could be used in factories to reduce the risk of harm of humans.

References

[1] David Whitney, *ROS Reality: A Virtual Reality Framework Using Consumer-Grade Hardware for ROS-Enabled Robots*, Brown University, Rhode Island.

[2] Nishanth Kumar, `https://github.com/h2r/ros_reality_tmp/tree/movo-VR`

[3] `https://velodynelidar.com/vlp-16.html`

[4] `http://wiki.ros.org/rosbridge_suite`

[5] `https://github.com/siemens/ros-sharp`

[6] *Implementing a Geometry shader for a pointcloud*, `https://answers.unity.com/questions/1437520/implementing-a-geometry-shader-for-a-pointcloud.html`