Remote Virtual Reality for Service Robots

Faizan Muhammad, Amel Hassan, Ahmed Ehab Gado

Abstract

This project aims to bridge the gap between a service robot's state and a human user. We believe that developing and using this technology would contribute to a deeper understanding of a robot's state during software and hardware development of these robots as well as be a potential platform for a range of different applications such as telepresence robots, remote robot tours, remote robot debugging and robot monitoring. For the scope of this project, we used a TurtleBot2 with the Oculus Rift virtual reality system. To connect these two systems, Unity was used as the middleware. Through the use of ROS Bridge and ROS_Sharp, we were able to depict continuous video streams of the robot's environment in the Oculus headset corresponding to the color and depth images. Currently, we are able to locally and remotely provide users with a bandwidth-efficient and realtime experience of a TurtleBot's environment.

Important Links:
Presentation: http://bit.ly/2Ifc1wy
Github repository: https://github.com/faizan-m/revrsr

1 Introduction

Being able to view and manipulate an environment from a remote location is the closest experience to teleportation one can achieve. In particular, robots can be teleoperated by humans to traverse through dangerous environments, such as war zones to perform military operations, or environments unsuitable for life, such as space. With decrease in prices of virtual reality systems (VR), using VR interfaces for robot teleoperation is becoming a more feasible and applicable feat. Such a method has the potential to provide its users with an immersive, detailed, and accurate replica of a robot's environment while ensuring their safety.

The main structure of our project revolved around the use of TurtleBot2 and the VR system, Oculus Rift. To render the TurtleBot's environment in the Oculus headset we used Unity, a game design platform, as the middleware. To connect ROS and Unity, we used ROS Bridge, a package and API which provides a JSON WebSocket interface to ROS topics. In addition, we used ROS_Sharp, a set open source software libraries and tools in C# that enables communication between ROS and Unity. Furthermore, we used a ROS package, web_video_server, to provide Unity with a video stream of ROS images that can be accessed via HTTP.

Our criteria for a succesful project were the following:

1. Visualize TurtleBot's environment in an Oculus headset remotely
2. Optimize accuracy and speed of transmitted images from ROS to Oculus
3. Improve user experience by minimizing the possibility of motion sickness

We followed an incremental appraoch outlined below and given the results of our project, we believe we have fulfilled the goals and expectations we set out with. The code for this project can be found in our GitHub repository.

2 Related Work

2.1 ROS and Unity Connectivity

The majority of researched projects opted to use the approach of using ROS and Unity, a game development platform, to display a robot's environment in virtual reality. These projects are listed in the Image Transportation section below (Section 2.2). To connect these two systems, RosBridge, a package and API, provides a JSON interface and WebSocket transport layer that allows non-ROS programs to subscribe and publish to ROS topics. Furthermore, there are several packages that connect ROS to Unity specifically, such as ROS_Sharp and the ROSBridge library. In a York University study, robot and virtual environment middleware was researched and tested in efforts to find an optimal solution [2]. In the paper From ROS to Unity, it is discussed that while there exists middleware for robots and virtual reality systems respectively, these provided software are not easy to integrate. The authors purpose Unity as the best middleware and they discuss their approach to linking ROS and Unity. In ROS, messages are passed using an internal protocol based on TCP/IP sockets, however it is much simpler to use the rosbridge framework and WebSocket as a communication layer as it allows two way communication with an external source. Many libraries exist linking Unity and the WebSocket protocol. Rosbridge communicates messages to and from the external world in the form of yaml. Sometimes these yaml strings can be used directly by the external word, however, it may be convenient to use JSON, which creates instances of objects using Unity.

2.2 Image Transportation

The most discernible variance among the researched projects was the method used to transport image data from ROS to Unity. When choosing a method, we had to consider speed of data transfer, accuracy, and quality. While we attempted several of the proposed approaches described below, in the end, we took a unique route by utilizing video streaming.

Mathias Thorstensen, a former master's graduate student at the University of OSLO in Robotics and Intelligent Systems, has done extensive work in visualization of robotic sensor data with augmented reality for his thesis in Spring 2017 [4]. He used the virtual reality platform, HTC Vive, and the game development platform, Unity to evaluate the effect augmented reality has on humanrobot understanding. To render images in VR, Thorstensen first attempted to directly receive point cloud data from ROS to Unity objects, however, due to their massive size, point clouds caused a bottleneck effect over the network. To counter the bandwidth issue, he resorted to using the compressed images from the topics /camera/rgb/image_raw/compressed and /depth/image_raw/compressed_image published by ROS. To access the individual depth values in Unity, he attempted to use Unity's built-in image load function, however, no image was created as the 8 byte PNG signature was missing. Although this issue was resolved by removing the first 12 bytes of the image, the image was still unusable because the compression was too strong. The dynamic gradient of the depth image was reduced to merely five different shades. Thorstensen then switched to using raw depth images from the /depth/image_raw topic published by ROS. While Thorstensen did not experience bandwidth issues, when implemented into our own project, we experienced significant lag in the rendering of the depth image. Thorstensen then reconstructed the point cloud in Unity by passing pixel coordinatedepth pairs through a function that calculated their corresponding 3D points. Translation of image coordinates to the normalized image plane, the camera's coordinate frame, was achieved by multiplying the image coordinates with the inverse calibration Matrix (Figure 2.1 and 2.2).

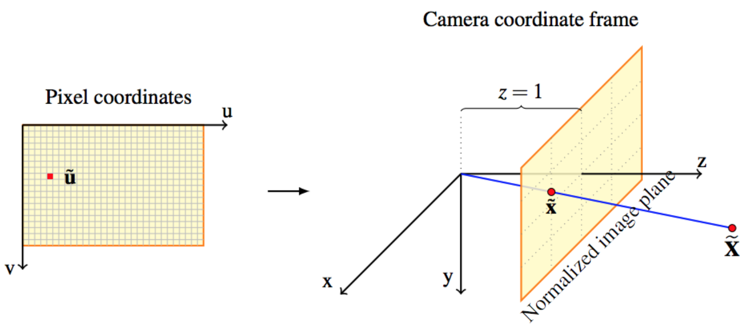
Figure 2.1 [4] Matrix Model - Image translation can be done by multiplying the image coordinates matrix, u, with the inverse of the calibration matrix, K, giving the corresponding point in the normalized image plane, x.

$$\tilde{\mathbf{x}} = K^{-1}\tilde{\mathbf{u}}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{f_u} & 0 & -\frac{c_u}{f_u} \\ 0 & \frac{1}{f_v} & -\frac{c_v}{f_v} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{u}{f_u} - \frac{c_u}{f_u} \\ \frac{v}{f_v} - \frac{c_v}{f_v} \\ 1 \end{bmatrix}$$

Figure 2.2 [4] The transformation from pixel coordinates and a depth measure to the corresponding 3D point



Once a new point cloud was constructed, the old one was deleted. Real time inspection showed that the frame rate was low, thus, even though image topics were used in an effort reduce bandwidth saturation, in the end, this method still resulted in latency issues.

A team from Brown University's Human to Robots lab created ROS reality, an over the Internet teleoperation interface between a ROS based robot and a VR headset. This same group performed research regarding robot teleoperation with VR and ROS reality [6]. Similar to the aforementioned research project, they used Unity and HTC Vive. Using the robot's URDF, they were able to build a local copy of the robot in Unity, which we have also incorporated into our project. Color and depth images were received from ROS and were then converted to a point cloud using Unity's custom shader. Although a public github repository is made available, we refrained from using ROS Reality because we could not depend on it being up to date. Instead, we used ROS_Sharp as it is company maintained and allows room for growth and scalability.

Another group of Brown University researchers, many of whom also worked on the ROS Reality system, worked on a project that also involved the combination of Virtual Reality and ROS [5]. Their project revolved around comparing the efficiency of different robot teleoperation methods in performing tasks such as cup-stacking. Similar to Thorstensen, this group of researchers used HTC Vive and Unity. The opted not to transfer point cloud data from ROS to Unity due to the latency issues that emerge. They used compressed depth and color images instead and used a custom GPU shader to reconstruct the point cloud on the VR computer. Unlike Thorstensen, they did not mention any issues with the quality of the compressed depth images. By the conclusion that the VR system maximized the user's success in teleoperating the robot to stack cups, it follows that the image quality of the robot's environment in HTC Vive must have been satisfactory. While this project shows that is possible to use compressed depth images, the explanation of their implementation was vague and thus unhelpful during the course of our project.

2.3 User Experience Considerations

A common concern when using VR systems is the side effect of motion sickness that comes with prolonged use. In our project it was important to incorporate elements that combat motion sickness because the usefulness of our system depends on the user's experience. If the user is incapacitated, they will not be able to reap the benefits of viewing a robot's environment from a remote location. In an article by Hettinger and Riccio, it is mentioned that empirical investigation on visually induced motion sickness (VIMS) is challenging because of the idiosyncratic nature of the sickness [3]. However, experiments that demonstrate relations between "vection" and sickness have shown that increasing the user's experience of self-motion, without actual physical displacement, and a high rate of optical flow in VR increase the chance of sickness. Motion sickness can be reduced by decreasing the intensity of pseudo self-motion by decreasing the speed at which visual information changes within the VR headset. In addition, having fixed points to focus the view on can also aid in reducing nausea.

In his Master's thesis, Daniel Bug uses Oculus Rift to control a mobile robot [1]. In his project, he implements visualization of the robot's environment through Oculus. In his paper, he explains how to compress point-cloud data into shapes to reduce the intensity of the image while maintaining its quality. In this method, during preprocessing, unnecessary points are filtered out. This works especially well for flat rectangular surfaces. Such an implementation reduces data traffic and allows images to smoothly be transmitted to Oculus, reducing the blur associated with rotating in Oculus, and thus reducing the chance of sickness.

While reducing the flow or sharpness of images is one way to combat motion sickness, it is important to realize that this may introduce other issues when dealing with robot teleoperation. For instance, if smoothness of VR is enforced, then there is no way to discern whether or not the robot's state is accurate and up to date. This can then lead to the user sending erroneous commands to the robot.

3 Methodology and Technical Details

3.1 Problem Formulation

We identified three main problems that need to be tackled to provide a good user experience and we began with outlining our strategy to overcome them.Firstly, Virtual Reality demands a consistent and high frame rate (around 90 Frames per Second) that cannot be guaranteed by a system operating wirelessly. Hence to ensure consistent streaming we decided that we would need a software system on client's end that is responsible for generating Oculus environment and interacting with the client directly instead of having the Turtlebot deal with it.

Secondly, even after such a software system removes a lot of pressure on Turtlebot's data transmission. There is still a great possibility that there are hiccups in the the bandwidth or latency due to the wireless network. Therefore, we decided that the 3D representation in the client's end of software will remain traversable with the latest possible data. This means that in the best case scenario client will notice no problems but in the worst case scenario, the client has to deal with a frozen scene within Virtual Reality rather than having the whole Virtual Reality experience frozen.

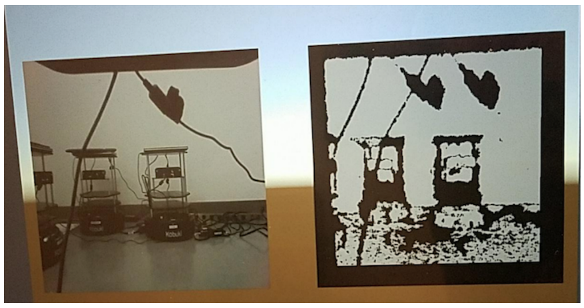
Finally, the Turtlebot can only see the environment well in its own height and relative to a human it is pretty short. This means that fully perceive the environment a user would have to crouch. We decided to scale the user's height in Virtual Reality such that the user is about the height of a Turtlebot. Not only does this give a clearer perspective of the data but it also helps the user relate and understand the robot's state better.

3.2 Initial Approaches

As described in the Related Works section, there were various approaches to rendering the robot's environment in Unity and Oculus. In the beginning phase of the project, we found an RViz Oculus plugin that theoretically was able to render an RViz view in the Oculus headset. Below is a sample display of how the scene would look in the Oculus format (figure 3.1.1). This would allow us to run our system solely on Ubuntu machines. We would have then used ROS_MASTER_URI to configure multiple machines to use a single master. However, since this package was last updated in August 2013, the required Oculus package versions were not available, only newer versions were downloadable. Thus, Oculus was not compatible with our Ubuntu machines and we could not use the Oculus RViz plugin. So instead we had to use one Ubuntu machine and one windows machine with Unity and ROS_Sharp.

Our next approach involved using one Ubuntu machine along with a windows machine. To transport images from ROS to Oculus, we used Unity, a game design platform, as the middleware. To connect ROS and Unity, we used ROS Bridge, a package and API which provides a JSON WebSocket interface. In addition, we used ROS_Sharp, a set open source software libraries and tools in C# that enables communication between ROS and application such as Unity. ROS_Sharp contains many sample scripts for use of ROSBridge. Our first attempt with this system setup was transporting compressed RGB and compressed depth images from ROS to Unity. Our goal was to decompress these images in Unity to reconstruct point cloud in Unity objects. However, we experienced difficulty with subscribing to the compressed depth image. When we subscribed to /depth/image_raw/compressed_image, no image was appearing in Unity. After further research, we figured out that ROS compresses depth images in png format by default. For this format to be interpreted by Unity, the image must be cut at a specific 8 byte signature location. In his Master's Theisis, Thorstensen mentions solving this issue by removing the first 12 bytes of the image. However, this presented yet another problem: the compression of the image was too strong, leaving the depth images to have gradients of merely five shades [4]. Thus, we tried using ROS's jpeg compression that is optimized for color images to compress the depth image. The image was able to be viewed in Unity. Unfortunately, in gradient images, this jpeg compression method oversimplifies the data and ruins the quality (Figure 3.2.1).

Figure 3.2.1 This is a display in Unity of images from ROS. The image to the right is a compressed RGB image from the ROS topic /depth/image_raw/compressed_image. While the image to the left is a compressed depth image in jpeg format.



Next, using the same ROS to Unity communication setup, we attempted to send compressed RGB images from the ROS topic /camera/rgb/image_raw/compressed but raw depth images from the topic /depth/image_raw. Although Unity was able to receive the images, there were latency issues due to the memory footprint of the raw depth image. Figure 3.1.2 shows an image of the rendered RGB image and the depth image appears in the corner. In Figure 3.1.3, the latency issues of the raw depth image are apparent: the red gradient depth image in the corner does not match the corresponding RGB image displayed. While updating the raw depth image took no more than a second at worst, we still wanted to improve our solution. To troubleshoot, we were going to reduce the size of the depth image, but then we discovered an even better solution: video streaming image content from ROS to Unity.

Figure 3.1.2 This is a sample display of compressed RGB images and raw depth images from ROS being rendered in Unity. The raw depth image, with red gradient at the top right corner, is in sync with the RGB image.

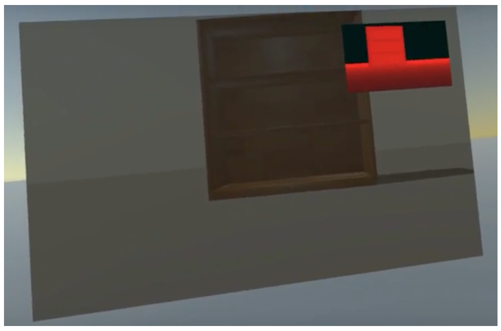
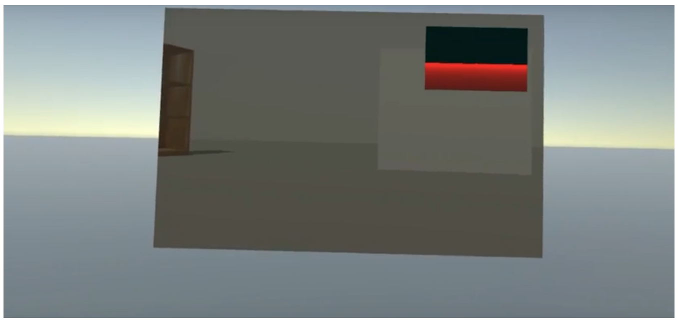


Figure 3.1.3 This is another sample display of compressed RGB images and raw depth images from ROS being rendered in Unity. However, the raw depth image, with red gradient at the top right corner, is in not in sync with the RGB image.



3.3 Final Approach

We used Web Video Server in ROS on Turtlebot to generate two video streams. One for the RGB image and the other for depth image. Then we modified the code we had to display these streams instead of pictures received frame by frame through ROS_Sharp. Since Unity is not compatible with Mjpeg video format, some external scripts were used to add that compatibility. This led to an instant increase in the quality and reliability of the transmission while decreasing our bandwidth. Moreover, the stream settings can be dynamically controlled from within Unity. Therefore it is possible to modify the video quality on the client end and thereby optimize for best experience based on the particular network infrastructure available. A schematic of the ReVRSR system is shown in Figure 3.3.1.

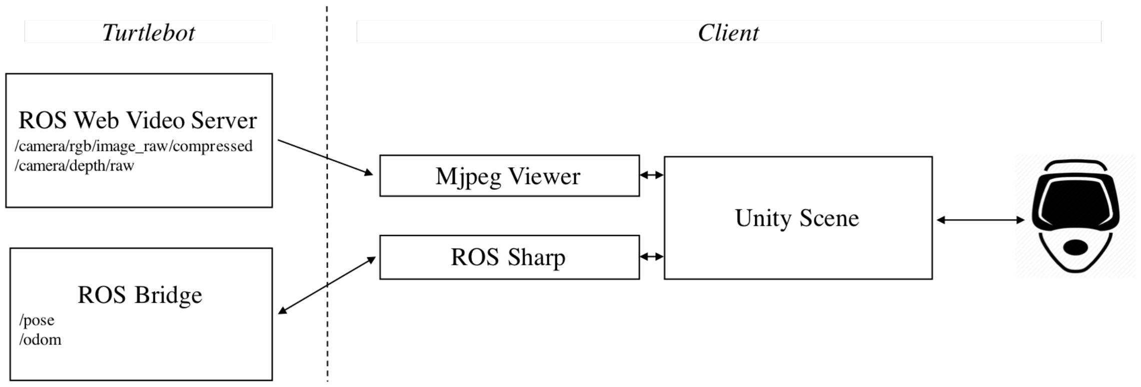


Figure 1: Schematic of ReVRSR system: ROS Web Video Server and ROS Bridge are ROS nodes that provide access to topic information. Mjpeg Viewer and ROS Sharp are Unity plugins that provide access to information about the robot and its surroundings. A Unity scene is built and shown to the user via an Oculus Rift.

4 Experiments and Results

4.1 Example Experience

To run the system, first the ReVRSR launch file needs to be launched from a TurtleBot which comes in the ROS package we provide on github. On client's end, we need to know the IP of the TurtleBot and that IP needs to be plugged in for ROS_Sharp and video streams. Then we can run the package in Unity which interacts with Oculus. The client can then put on the Oculus headset to be immersed in a replica of the TurtleBot's environment. The user can view a continuous video stream of images from ROS without any latency issues nor jarring image updates. Below are images of what the user experiences when using this system (Figures 4.1.1, 4.1.2, 4.1.3).

Figures 4.1 - These are depictions of what the user experiences in our virtual reality system using the Oculus headset

Figure 4.1.1

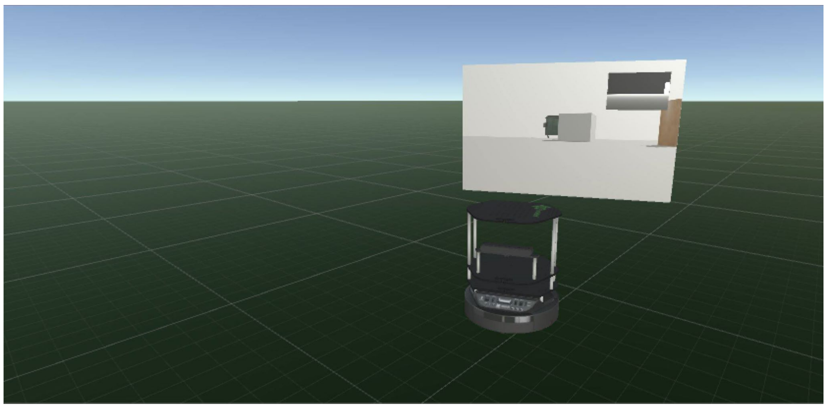


Figure 4.1.2

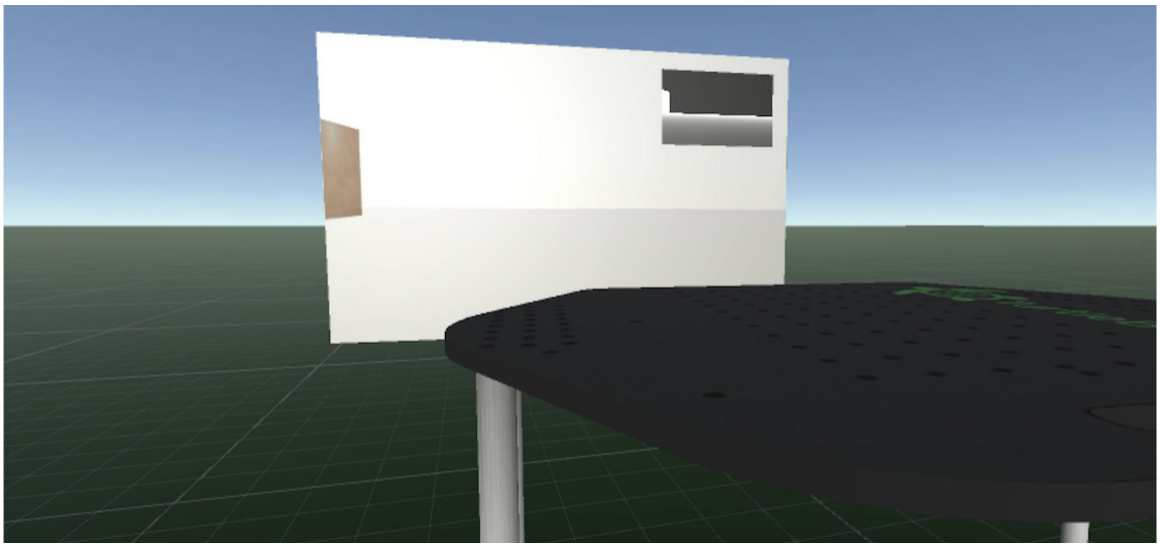
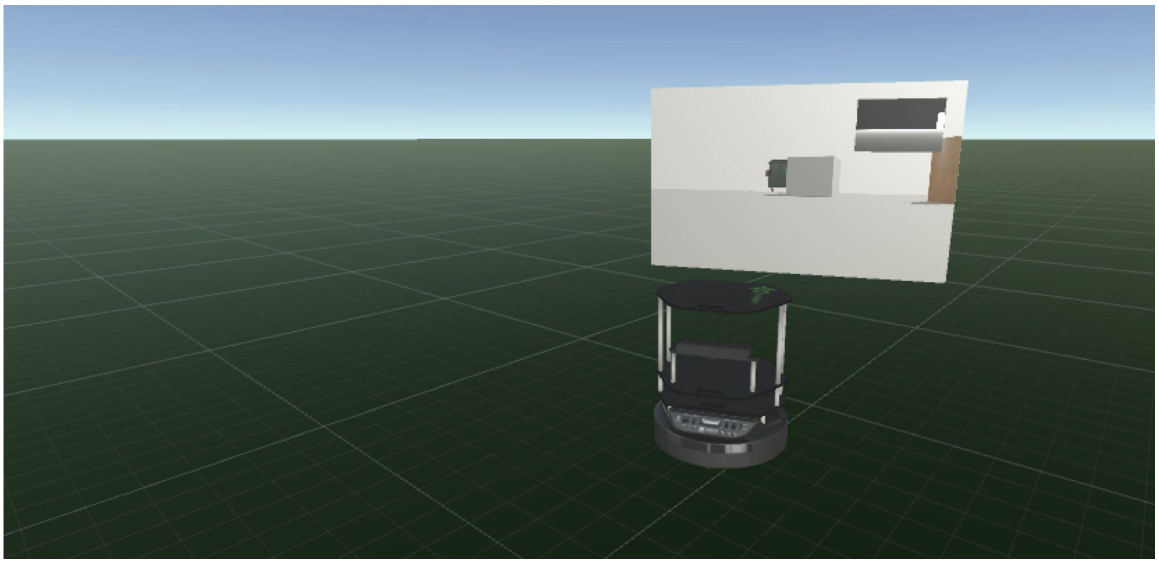


Figure 4.1.3



The client can also use the Oculus Touch controllers to turn the Turtlebot around and make it go forward and backward. The user can traverse through the robot's environment and control it remotely as two different machines are being used. A demo video can be found on the GitHub repository for this project.

4.2 Discussion

The final approach of using video streaming for our project provides many advantages over our initial approach as well as methods suggested by our research. For instance, our approach does not suffer from latency or bandwidth issues; we are able to render images in real-time. In his Master thesis, Thorstensen used the approach of reconstructing compressed RGB images and raw depth images into a point cloud in Unity[4]. Once a new point cloud was constructed, the old one was deleted. Real time inspection showed that the frame rate was low, thus, even though image topics were used in an effort reduce bandwidth saturation, in the end, this method still resulted in latency issues. In addition, unlike other similar projects, we did not use image compression nor conversions, ensuring that the accuracy of the images were preserved.

One of the issues in our final approach is that since we are using a video stream, it is more difficult to generate point clouds because the video streams need to be broken down into individual frames before we can use them to create point clouds. Moreover, one of the limitations of this project is that the data we can display in Virtual Reality is limited to pose of the robot along with the color and depth images.

Moreover, another limitation we observed is that there is a small cumulative error in odometry since Unity tries to recreate pose through odometry and if there are a lot of fast rotations involved then it loses track of the angular orientation. However, we observed this in Gazebo simulation where we rotated the TurtleBot at unrealistic speeds. Hence in actual usage this would not be significant enough error.

4.3 Evaluation

As listed in the introduction, our goals and criteria for a successful project were as follows:

1. Visualize TurtleBot's environment in an Oculus headset remotely
2. Optimize accuracy and speed of transmitted images from ROS to Oculus
3. Improve user experience by minimizing the possibility of motion sickness

All of our main goals were accomplished. That is, we were able to provide our user's with the ability to experience a robot's environment remotely in virtual reality. Relative to other projects, our image stream provides real-time and low bandwidth performance. In addition, the user's that experienced the virtual reality environment did not experience motion sickness. Furthermore, even though we researched similar projects, we were able to construct a novel solution by video streaming both RGB images and depth images. Although we were not able to render a 3D point cloud, this was a stretch goal, and our system provides a strong foundation for proceeding in this direction.

5 Conclusion and Future Work

Our project provides a foundation for building even more sophisticated replicas of a robot's environment in a virtual reality system. In the future, we hope to use the video stream to reconstruct a point cloud in Unity to provide the user with a 3D experience of the robot's environment. In a 3D experience, however, there would be no constant horizon. This increases the risk of motion sickness because the user loses the ability to use the horizon as a visual anchor and break. Since video streaming allows us to transmit images without latency issues, constructing a point cloud will not result in a discrepancy between what the robot is actually seeing and what the user is seeing. That is, the point cloud will be constructed quickly enough for their views to stay in sync.

Furthermore, we hope to improve and expand on the user experience by including an interface in the Oculus system. For instance, there could be a drop down menu where the user can choose to view a 3D experience, a 2D video stream, or a raw depth image. This allows our product to be versatile as it takes into consideration what information the user needs to access as well as the intensity of virtual reality experience they can handle. In addition, we would like to incorporate a map interface where the user can specify a navigation goal for the robot to traverse to.

References

[1] Daniel Bug. Oculus rift control of a mobile robot. Master's thesis, KTH Computer Science and Communication, 2017.

[2] R. Codd-Downey, Mojiri P. Forooshani, A. Speers, M. H. Wang, and Jenkin. From ros to unity: Leveraging robot and virtual environment middleware for immersive teleoperation. In *Information and Automation (ICIA), 2014 IEEE International Conference on*, pages 932–936. IEEE, 2014.

[3] Lawrence J Hettinger and Gary E Riccio. Visually induced motion sickness in virtual environments. *Presence: Teleoperators & Virtual Environments*, 1(3):306–310, 1992.

[4] Mathias Ciarlo Thorstensen. Visualization of robotic sensor data with augmented reality. Master's thesis, York University.

[5] David Whitney, Eric Rosen, Elizabeth Phillips, George Konidaris, and Stefanie Tellex. Comparing robot grasping teleoperation across desktop and virtual reality with ros reality.

[6] David Whitney, Eric Rosen, Elizabeth Phillips, Daniel Ullman, and Stefanie Tellex. Testing robot teleoperation using a virtual reality interface with ros reality.