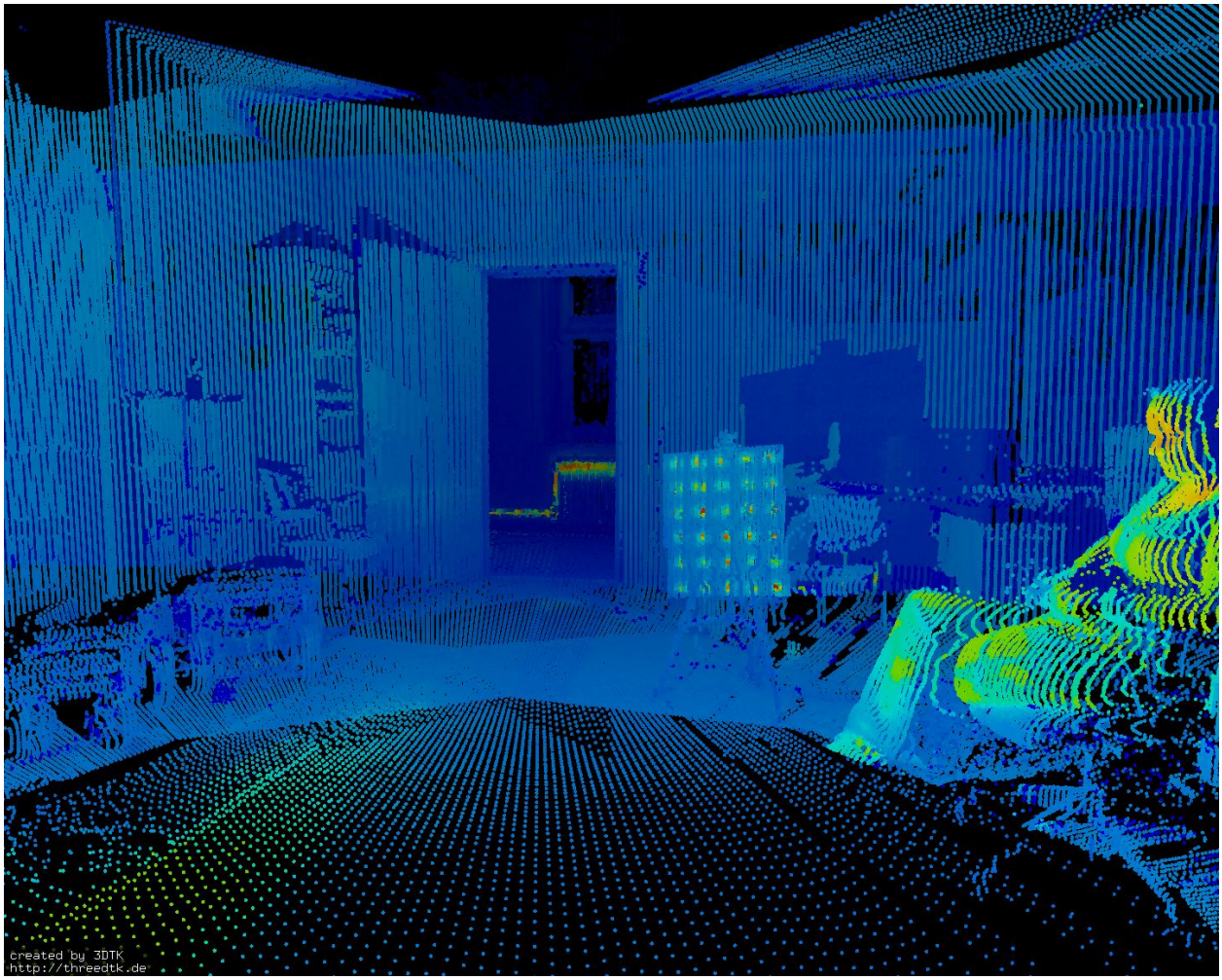
REAL-TIME POINT CLOUD CONSTRUCTION AND INFORMATION SYSTEM

Project Report



Authors: Willem Lefferts, David Heetebrij,
Andrei Martashov, Stan Smeitink
Educational institution: Saxion
Client: Research group of ambient intelligence
Mentor Saxion: Alejandro Moreno Celleri
Mentor Research group: Danny Plass

Table of content

List of Abbreviations

ROS: Robot Operating System. The software used to control and extract data from the robot.
VR: Virtual reality. VR is a simulated experience that can be similar to or completely different from the real world.
JSON: JavaScript Object Notation, is an open standard file format, and data interchange format.
API: Application Programming Interface (API) is a computing interface which defines interactions between multiple software intermediaries.

1. Introduction

The Dutch fire service does many tasks besides the obvious tasks like extinguishing fires, such as providing fire safety information, rescuing people and animals, assisting in accidents with hazardous substances or trapped people and disaster management. From all these tasks, however, one of the most risk prone is that of fires and accidents in difficult to reach places or large buildings. One of the biggest problems in most of these situations is the lack of the current information about the situation. This lack of information makes it a lot more difficult for the firemen and especially their commanders to make proper, well thought-out, on the spot decisions. But in today's day and age there surely must be some sort of machine that can help the firemen in these difficult situations.

That is why the research group Ambient Intelligence, together with the firemen, have been working on a solution to this problem. What they have come up with is a robot that will be able to enter these dangerous and difficult to reach places without having to risk the safety of a fireman. This robot will be armed with cameras that will send their data back to the firemen so they will have real-time visual information. Thanks to this information they can then make more appropriate choices in order to combat the fire.

The part we will play in this whole project is making sure the firemen will be able to actually see and use the information provided by the robot. To achieve this task we will create a connection between the robot’s cameras and a VR headset that the firemen will use to see a visualisation of the data gathered by the robot. This means we are responsible for establishing a connection between the source of the robot’s data and the VR headset. The data will be used to create a point cloud. A point cloud is a set of data points in space. This will give the firemen a clear indication of the environment around the robot and still be able to distinguish features of a building when it is on fire and normal visibility would be compromised.

2. General problem analysis

The situation at the beginning of this project is as follows:

- The mechatronica team is developing a stereo camera that will be mounted on the robot and will generate the necessary data.
- This camera will be turned into a point cloud. This has been established in an earlier project that worked on this problem. The reason why the camera data will be turned into a point cloud is because these will give you a good understanding of a burning and smoke-filled environment.
- The firemen want to use VR to view these point clouds.

This means we will need to develop the following according to the client's wishes:

- A way of connecting the source of the data with Unity.
- Display the point cloud inside the VR headset.
- Optimizing the data so that the VR application will run smoothly.

For all of these points we will need to conduct both qualitative research and experimental research. We will need to do this in order to find out what in theory is the optimal solution for this problem and to then find out if it actually works.

3. Connecting ROS and Unity

Transferring data from ROS to Unity

Background

At the very beginning of this project, our client connected us with Gerjen ter Maat. Gerjen is a researcher with Mechatronica research group and specializes in robots and operating robots. He also works on the robot's side of this project which meant we had to coordinate with him how we were planning on establishing the connection between Unity and the robot. Because all of us had never worked with robots before we did not know where to start. Luckily Gerjen told us about ROS, or Robot Operating System. This robotics middleware is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms[1].

After some time researching ROS and following some Mechatronica workshops about ROS we had a pretty clear idea of how it worked, so we started asking Gerjen more questions about the specifics of the data our side of the project would receive from the robot. He then provided us with a ROSbag. A ROSbag is a file format in ROS for storing ROS message data. Bags -- so named because of their .bag extension -- have an important role in ROS, and a variety of tools have been written to allow you to store, process, analyze, and visualize them[2]. We assumed for the rest of the project that the ROSbag is what the data looks like. The only real difference with the eventual final product is that a ROSbag is a recording and the robot will send the data real-time, however, the data it will send is basically the same.

To get a point cloud out of a ROSbag we also received a stereo_image_proc launch file from Gerjen. When this file is run from within a catkin package with the stereo_image_proc dependency, while at the same time playing a compatible ROSbag, it creates point cloud data. This data can then be used by renderers and other visualizers, such as RViz, to display a point cloud[3].

What is also important to notice, is that the way we can send and receive data from a ROSbag should be the same as real-time streaming. The transfer of data is managed using ROS topics. Topics in ROS are named buses over which nodes exchange messages. Topics have anonymous publish/subscribe semantics, which decouples the production of information from its consumption. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic; nodes that generate data publish to the relevant topic. There can be multiple publishers and subscribers to a topic[4].

During our research we were also looking for ways to actually send the data of these ROSbags. Because we already knew that the recipient had to be Unity, we could exclude a lot of methods,

---

[1] "About ROS - ROS.org." https://www.ros.org/about-ros/.
[2] "Bags - ROS Wiki - ROS.org." http://wiki.ros.org/Bags.
[3] "stereo_image_proc - ROS Wiki." 18 Jul. 2016, http://wiki.ros.org/stereo_image_proc.
[4] "Topics - ROS Wiki." 20 Feb. 2019, http://wiki.ros.org/Topics.

simply because they did not work with Unity. We quickly found a GitHub repository called ROS#, which advertised itself as a set of open source software libraries and tools in C# for communicating with ROS from .NET applications, in particular Unity[5]. This sounded like it was exactly what we were looking for, so we immediately started experimenting with it by doing the tutorials.

During these experiments we got aware of another piece of software that showed great potential to what we were trying to achieve. The ROSbridge got introduced to us in the early steps of the tutorials. ROSbridge provides a JSON API to ROS functionality for non-ROS programs. There are a variety of front ends that interface with rosbridge, including a WebSocket server for web browsers to interact with[6]. ROS# uses this WebSocket connection in one of it's tutorials[7] for sending data from ROS to Unity, so we wanted to try and use that as well.

State of the art

The state of the art regarding this specific topic is pretty difficult to define. Since it is such a niche topic there are very few to no implementations that are likewise to ours. Libraries that try to implement the features we were after are the closest thing that come to state of the art. ROS# is most likely the best and most advanced in this regard which would make it state of the art.

Methods

This research, for the biggest part, is conducted by prototyping. This means that there is a lot of time during which we worked on this project that did not produce anything that we could use for this project's product. What it did produce was more knowledge on what we can not use. We will discuss these failed experiments in the results part of this chapter. Another downside of conducting our research this way, is that we will have less academic research to write about. Because of this, this report might look like we did not do that much considering how long we have been working on this project.

Results

Because of our research methodology, it is best to split up this section of the report into two. Namely what worked and what did not work. We will discuss the parts that did not work first.
As we already mentioned, during our research we discovered ROS# and ROSbridge. The experiments were mostly focussed on these two pieces of software. We started of by trying to complete the tutorials posted on the GitHub repository of ROS#. These led us to learn more about Catkin workspaces and packages. This topic is a pretty complicated one and the help of Gerjen during this phase of the project was absolutely critical. Setting up a workspace with a

[5] "siemens/ros-sharp: ROS# is a set of open ...." https://github.com/siemens/ros-sharp.
[6] "rosbridge_suite - ROS Wiki." 23 Oct. 2017, http://wiki.ros.org/rosbridge_suite.
[7] "User_App_ROS_GazeboSimulationExample."
https://github.com/siemens/ros-sharp/wiki/User_App_ROS_GazeboSimulationExample.

package including the specific dependencies one needs is a pretty precise task and it is easy to do a part of it wrong, resulting in the package not working. We noticed this was a pretty complex task since we did it wrong a couple times before doing it right. The complexity of this task can read about in the attachments of this document. So that was the first part where we lost a bit of time, but fortunately we did get the workspace up and running. With this workspace we could now visualize the point cloud inside ROS by using RViz, a 3D visualization tool for ROS[8].
The next task was streaming the point cloud data from ROS to Unity. For this we will use ROSbridge. Setting up the connection between ROS and Unity using the WebSocket provided by this library was fairly easy. Both the console logs of the ROSbridge as the one in Unity showed that a connection was successfully made. However, when the connection was made, Unity was supposed to subscribe to a ROS topic using a ROS# script, which it did not do.
At this point we started to get really stuck and there was very little progress. There were a number of things we tried to help resolve the situation but none of them worked. Some of the things we tried were:

- Use different libraries than ROS#
- Use Java or JavaScript to write a short program that forward the data to Unity
- Debugging the subscriber script to no end.

Whatever we tried it did not seem to work. One of the weird things was that the JavaScript we had written did subscribe to the topic and also received data when we sent some. This meant that the problem was somewhere inside the Unity project. But because we could not find the problem we decided to send an email to Yvens Rebouças Serpa in the hope that he perhaps could help us overcome this obstacle. He suggested a number of things, one of which was to retry the process in a new Unity project. We had already tried this before and this seemed too simple a fix. But, since we could not think of another fix we decided to just try it and see if it might work anyway, which it somehow did. The first project must have had something wrong inside the Unity settings or something might have gone wrong importing the scripts into the assets. What exactly went wrong we do not know, but it eventually worked. We also replicated the working project in a different project to see if it would work again and it did.
It is unfortunate that we lost quite a lot of time on something like this, but it does work now. We first tested the connection and subscription by publishing the image instead of point cloud data, since ROS# does not have a point cloud subscription script. These images got across without any trouble.
Earlier during the project we had already looked into scripts that could subscribe to point cloud data topics, now it was time to see which one worked. Because we had looked into this subject earlier there were some libraries that looked very promising. The one we tried first was one of those promising libraries: unity_assets/PointCloudStreaming made by Immo Jang on GitHub[9]. This library is an extension specifically made for ROS# to subscribe to point cloud data. It includes a subscriber, renderer and a OpenGL extension for the camera inside Unity. Setting everything up so this library works as it is supposed was not very difficult. When running the project it connected without any issues to the ROSbridge and also subscribed to the correct

[8] "rviz - ROS Wiki." http://wiki.ros.org/rviz.
[9] "inmo-jang/unity_assets - GitHub." https://github.com/inmo-jang/unity_assets.

topic. Publishing the point cloud data from ROS onto the ROSbridge also went without a hitch. After a short delay, Unity starts rendering and showing the point cloud.
So, finally, after a lot of trouble the connection between ROS and Unity was working. In the attachments is a document that can give a more detailed look into how the connection works and what we specifically had to do to get it to work.

Conclusions and discussion

To summarize all of the results of our experiments: ROS# in combination with ROSbridge and the point cloud subscriber works very well and is easy to set up, granted you do not make any mistakes. Unfortunately we cannot say too much about what definitely does not work because we are not sure how much impact the Unity project had on the results, but our guess is that it did have a lot of impact. What we can say for sure is that ROS# is easy and effective, especially if you know how to use ROS properly.
One thing we are not sure about is how flexible the connection is. We have only tested it with the following configuration:

- A PC with Unity
- A Ubuntu virtual machine that runs on the same PC
- The data is pre-downloaded on the Ubuntu machine

When the ROS side eventually gets from the robot, the robot will need to maintain connection with the VR headset or the machine on which the VR application is run. What we have tested is connecting the VR headset to a hotspot hosted by the PC. In this situation the headset and ROS establish a connection and the headset also receives data from ROS. However, this connection will weaken or disconnect when the headset gets too far from the machine running ROS due to WiFi having a limited range. This is probably going to be an important issue that will require extra research and experimenting when this research gets continued.

Recommendations

While we use ROSbridge with a WebSocket connection, ROSbridge itself is not tied to any particular transport layer. It might be possible to add a ROSbridge TCP package or node, for example, that communicates with ROSbridge using TCP sockets. This way you do not need to have both the machines on the same local network and they can have some distances in between them.
We also want to encourage more in depth research into handling an actual real time stream instead of pre recorded data. We do not expect any issues with connecting and getting data across. However, at that point it would be necessary that the data builds up the point cloud from the data it gets. Right now it is more similar to a video and the point cloud does not increase in size. This should be different in the eventual application.

4. VR Integration

With gesture control

Background

The Scarab “a robot” will be used by the firefighters for scouting and scanning hazardous areas. [10] At the moment the vision of the robot is really poor. The firefighters cannot guide the robot without standing physically next to it. They want to create a VR environment which gives you a clear indication where the robot is and also see what it sees. To do this the mecatronica team is working on a camera that scans the environment with thermal vision to create a point cloud. This camera will be attached to the robot later on. The data that the camera gathers should be visualized in a VR application that can run on the Oculus Quest.
For an even better idea of what is happening around the robot we thought of using gesture control to switch between point of view. This is because the robot is controlled by a big controller and they can not wield the Quest controllers.

State of the art

At the beginning of this project nothing really was done or researched for the VR part. There was only the idea to stream point cloud data and viewing it in VR using the Oculus Quest. There were a few videos online of people also trying this,[11] but no public projects. So we had to start from scratch and see if ROS# could also stream to the Quest.

Methods

For the VR part most of the research is done by trial and error and following many tutorials on the internet. At the very beginning of this project we did not have any data to work with. The mecatronica team was working on this. So to begin with we started to look into how to setup a VR application. Since Unity is a good platform to do this with and two of our group members already worked with it we decided to do it in Unity. Since VR is a very innovative subject there were a lot of updates along this project. So some of the code, plugins and assets we used earlier changed or had to be changed as well. Another difficulty was that there isn’t always a clear way to create some features simply because they weren’t done before. Then we had no choice to experiment and combine things we found. For the VR integration we made use of the

[10] “Scarab TX Firefighting Robot” 24 June 2019, https://www.fireproductsearch.com/scarab-tx-firefighting-robot/
[11] “Real-time Point Cloud Streaming to Oculus Quest” 03 June 2019, https://www.youtube.com/watch?v=jUlgNf5MMSA

Oculus Integration package in the Unity asset store.[12] This provided us with some nifty game objects, scripts and meshes. Later in the project it became more clear what the data will look like, so we could focus more on optimization.

Results

To create the VR application in Unity was a good choice because there were a lot of tutorials online on how to do this. The first prototype we made was just a simple application that lets us walk through a point cloud. This was done to see if there were any difficulties or high latency. Because the point cloud was quite small, this was not the case. It was not easy to find point cloud files that you could implement in Unity. Because of this we did not look for a bigger point cloud. Another reason we did not do this is that these files consist of one big mesh. This will always have a better performance than a point cloud that consists of individual points. Although we were no longer going to test with these files, we did get the idea to mesh the incoming point cloud data like these files.
Because the streaming from ROS to Unity took longer to accomplish we did not have any point cloud data to work with yet. So for the optimization part we used or created data that we thought was going to be almost similar to the real data. The Gesture control will work even if other parts of the application are not finished yet. We found a nice tutorial on Youtube on how to do this.[13]
Because the final product of this whole project is not finished yet there is a developer option so that the developer still can capture and save more gestures. The gestures that we put in the application now are able to change some things in the scene. If the next mecatronica is capable of getting the position of the robot relative to the point cloud then they can use these gestures to change the robot's point of view.
Near the end of the project streaming the point cloud between ROS and Unity was accomplished, but this only worked on the computer itself. We still needed to get this to work on the Oculus Quest. We managed to get the connection between ROS and the Quest, it also is receiving data, but the rendering isn't finished yet.

Conclusions and discussion

In conclusion, optimizing the point cloud for the Oculus Quest is still an open issue. We established the connection too late to propper test the point cloud in VR. We have some optimization done but this is only tested on PC and also not tested with the data stream.
Gesture control is working nicely, if the threshold is not set too high. Else it will always detect a gesture even when none is made.
The connection between ROS and the Oculus Quest seems to work. ROS shows there is a device connected and we added a console in VR to show the output of the ROSConnector. It is telling there is a connection and also that it is receiving data.

---

[12] "Oculus Integration" 27 May 2020, https://assetstore.unity.com/packages/tools/integration/oculus-integration-82022
[13] "Hand Tracking Gesture Detection - Unity Oculus Quest Tutorial" 02 March 2020, https://www.youtube.com/watch?v=IBzwUKQ3tbw

Recommendations

Because we did not establish a proper point cloud stream to the Oculus Quest there is a lot of testing to do. The first thing to do is to see if the data is coming in properly. The application is receiving data but the rendering failed. This is because the rendering part is created to render the point cloud in Unity on a PC. We thought it also would work for the Quest but this might not the case. More research is required to find out what the cause of this problem is. The second thing to do is, if the rendering is working it might be too much data for the Quest to handle and it will crash the application. So our or new optimization methods should be implemented.

5. Point Cloud Optimization

Background

For the interpretation of the data we receive from the robot's camera we are using a point cloud. A point cloud is a large collection of 3D coordinates, each visualized as a so-called point, which together give the appearance of three-dimensional shape.
Keeping track of so many individual points at a time requires a lot of computing power, therefore a method of optimization is needed to achieve an acceptable level of performance while having limited resources.

State of the art

Due to the general high resource demand of point clouds, they tend to be optimized wherever they can. Unfortunately, this usually means they are saved, or 'baked' as a single mesh, a single object. While this is fine in most cases, for this project we will be constantly supplied point cloud data, and will thus have to process the data in real time.

Methods

There are multiple ways in which optimization can be approached. For this project specifically we looked into distance optimization and mesh optimization.

Distance optimization was the first method that was looked into. The core concept is reducing the amount of points based on how far away they are from the camera, thus reducing resource costs.
This method was meant to be efficient while iterating through thousands of objects, without adding them to lists or arrays, since their number would affect the performance greatly. One of the first designs that was tested used box collisions to detect the objects within them, adding them to several lists and then iterating through those lists to destroy the objects. However, this didn't prove to be very efficient, since checking for collisions for the large number of objects cost a lot of time. Thus, it was reworked into the current method that only iterates through the objects once.

Mesh optimization was stumbled upon during research on different methods of distance optimization, specifically in Unity. We found an old free-to-use plugin made for Unity 4[14]. This plugin read a point cloud file from disk, iterated through all the points, grouped them into

[14] Llorach, G. (2018). *Point Cloud Free Viewer* [Unity Plugin]. Retrieved from https://assetstore.unity.com/packages/tools/utilities/point-cloud-free-viewer-19811

smaller, separate meshes and then saved it as a unity prefab. It would then load this prefab back into unity for display. This method was remarkably quick, but it came with a few problems.

Firstly, the plugin did not work outside of Unity's Editor mode due to the functions it utilized, and would thus not work in a finished product.
Secondly, it relied on hardcoded file paths, which would not work on different operating systems such as Android, which is what the Oculus Quest runs on.
Lastly, the plugin works exclusively with files in a certain file format that are read from disk.In our use-case we get the data streamed in live from sensors, while at the same time we don't know what formatting will be used.

Results

For the distance optimization, a prototype C# script was created for use in Unity. It is attached to the parent of all the objects that need to be optimized. It iterates through all the objects once, checks each object's position and compares it to the camera's position and determines the group that object is supposed to be in with a linear interpolation.
After that it calculates the culling factor which determines how many objects are going to be destroyed in the group. Culling factor takes the index of the group as a base and then either multiplies it by a factor or raises it to the power of the factor, depending on what settings are enabled.
While this script works only with objects in Unity, the method itself can be used in other various situations with small adjustments.

For the mesh optimization, as of writing this report two out of the three problems mentioned have been solved. A new C# script has been created, based on the original plugin. It is more generic that the plugin, in that it doesn't require adherence to a certain file format. By simply passing it coordinate data, and optionally colour data, it will group points together based upon settings, turn each group into a mesh, and then load the mesh into the scene. This solves the problem of file formats as a different interpreter scripts can simply utilize this script to load the data into the scene. At the same time this also solves the live-data problem as the script will simply turn whatever amount of points into a mesh, up to a certain point due to internal limitations in Unity, or create multiple meshes if the amount surpasses this limit. It also does not use any UnityEditor-specific functions, so it will work in a compiled product.
The final, remaining issue has not been fixed as of writing. Due to the decoupling and adapting of the original plugin's functions the script described above is not able to load a point cloud data file from disk. While relatively a minor point, as this function is not required for this project, testing the script is hard without some form of data to load, in the absence of a live data stream.
A second script which utilizes the first has been made to address this issue, but currently still suffers from the fact that it exclusively works on PC, and does compile into a build product.

Conclusions and discussion

Point Cloud data is hard on a system's performance. We have tried to reduce this impact, and have succeeded in coming closer to a finalized product. However, due to the nature of the target platform which has limited capabilities, and also requiring it to be run in virtual reality, these methods, while allowing for a significant increase in performance, may not be enough.

Still, we believe that at the very least we have provided future project teams a sizable foundation upon which to expand, and improve, so that an acceptable product can be reached.

Recommendations

While not specifically mentioned before, during the research phase we discovered one of the better ways to pass on point cloud data is by using bytes instead of strings. As this allows you to bypass the large performance impact from using the "string.Split" function.

6. Attachments

ROSbridge documentation - How to set up a connection between Unity and ROS

Visualizing the point cloud

To complete the steps described in this document you will need to have a (virtual) machine installed with Ubuntu 18.04 and ROS Melodic. As well as a windows machine with Unity installed on it.

An image for Ubuntu 18.04 can be found here:

https://releases.ubuntu.com/18.04/

And a tutorial on how to install ROS here:

http://wiki.ros.org/melodic/Installation/Ubuntu

From the Mechatronica team we have received a ROSbag which contains data that can be turned into a point cloud. To do this we also received a file called `stereo_image_proc.launch`. This file performs the duties of image_proc for both cameras, undistorting and colorizing the raw images. Note that for properly calibrated stereo cameras, undistortion is actually combined with rectification, transforming the images so that their scanlines line up for fast stereo processing. In this case we will use it to determine the positions of the points of our point cloud.

In order to do this we need to make a catkin workspace in our Ubuntu environment. A catkin workspace is a folder where you modify, build, and install catkin packages. A catkin workspace can contain up to four different spaces which each serve a different role in the software development process. The **source space** contains the source code of catkin packages. This is where you can extract/checkout/clone source code for the packages you want to build. Each folder within the source space contains one or more catkin packages. The **build space** is where CMake is invoked to build the catkin packages in the source space. CMake and catkin keep their cache information and other intermediate files here. The build space does not have to be contained within the workspace nor does it have to be outside of the source space, but this is recommended. The **development space** (or devel space) is where built targets are placed prior to being installed. The way targets are organized in the devel space is the same as their layout when they are installed. This provides a useful testing and development environment which does not require invoking the installation step. When ever referring to a folder which can either be a development space or an install space the generic term **result space** is used. We can make a catkin workspace by simply running the following commands:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
```

Additionally, if you look in your current directory you should now have a 'build' and 'devel' folder. Inside the 'devel' folder you can see that there are now several setup.*sh files. Sourcing any of these files will overlay this workspace on top of your environment.

```
$ source devel/setup.bash
```

To make sure your workspace is properly overlayed by the setup script, make sure the ROS_PACKAGE_PATH environment variable includes the directory you're in.

```
$ echo $ROS_PACKAGE_PATH
/home/youruser/catkin_ws/src:/opt/ros/melodic/share
```

Next up, to use this workspace we need to create a package. For a package to be considered a catkin package it must meet a few requirements:

- The package must contain a catkin compliant package.xml file.
    - That package.xml file provides meta information about the package.
- The package must contain a CMakeLists.txt which uses catkin.
    - If it is a catkin metapackage it must have the relevant boilerplate CMakeLists.txt file.
- Each package must have its own folder
    - This means no nested packages nor multiple packages sharing the same directory.

The simplest possible package might have a structure which looks like this:

```
my_package/
  CMakeLists.txt
  package.xml
```

To create a new package, first we need to change to the source space directory of the catkin workspace:

```
$ cd ~/catkin_ws/src
```

Now use the catkin_create_pkg script to create a new package, you can also add dependencies to the package at this step. We will add the stereo_image_proc dependency. This command looks like this:

```
$ catkin_create_pkg pointcloud_visualizer std_msgs rospy roscpp
stereo_image_proc
```

Now that we have created the package we cd back to the workspace root folder so that we can remake the workspace with the package:

```
$ cd ~/catkin_ws
$ catkin_make
```

We recommend creating a new folder inside the package directory called launch, in this folder we will deposit the ROSbag file as well as the stereo_image_proc.launch file. Now we are ready to visualize the point cloud from the ROSbag.
We need to start with running ROScore by running the following command:

```
$ roscore
```

This step is not necessary when the ROSbridge has already been launched.
Then, open RViz by running the following command:

```
$ rosrun rviz rviz
```

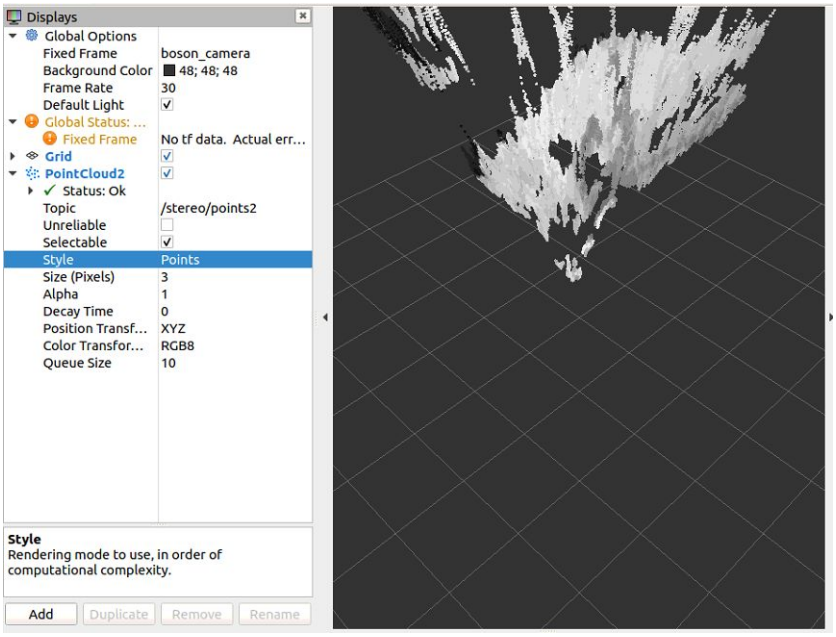
Then, in a separate terminal run the stereo_image_proc from the package. You can do this as such:

```
$ cd ~/catkin_ws/src/pointcloud_visualizer
$ roslaunch pointcloud_visualizer stereo_image_proc.launch
```

Be sure that you sourced the setup.bash of your workspace, otherwise you will not be able to use the package or the launch file.
And lastly, we can run the ROSbag. We need to do this from a different terminal as well:

```
$ rosbag play [name of rosbag]
```

Now we need to open up RViz and click the “add” button on the lower left hand corner of the screen. Click on the tab “by topic” and scroll down to the topic stereo/points2, and add “PointCloud2”. Now, the last thing we need to do is change the “Fixed Frame”, which you can find in the upper left hand corner of the screen. In this example we need to change it to “boson_camera”, but this can change depending on what ROSbag you are using. At this point RViz should visualize the point cloud, which looks like this:



Sources:
http://wiki.ros.org/stereo_image_proc
http://wiki.ros.org/catkin/workspaces
http://wiki.ros.org/catkin/Tutorials/create_a_workspace
http://wiki.ros.org/ROS/Tutorials/CreatingPackage

Streaming Data

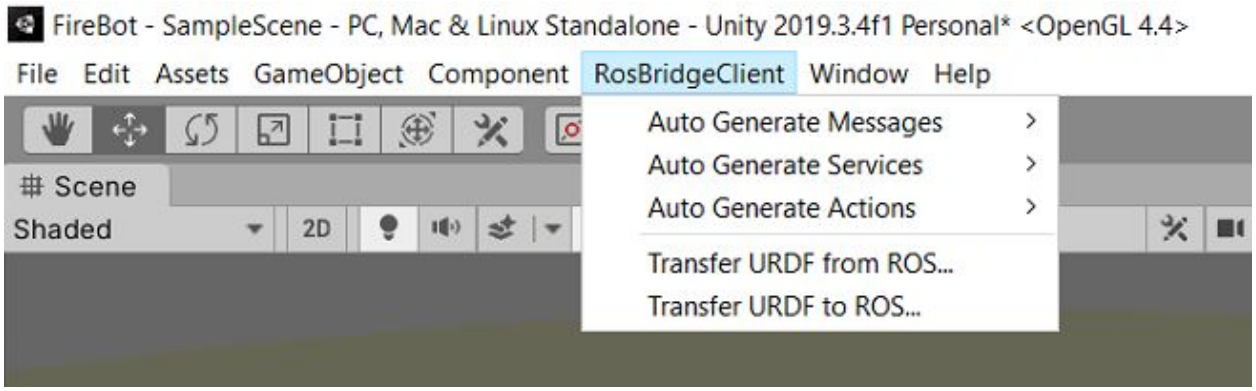
To start streaming data from ROS to Unity, we will need a connection between the two. For this we will use ROS#. ROS# is a set of open source software libraries and tools in C# for communicating with ROS from .NET applications. ROS# has an extensive tutorial on how to install all the necessary software needed to use ROS# that can be found on their repository page. However, for what we want to use ROS# for we don't need to do all of these tutorials. Instead we can limit ourselves to only a couple of steps.

First of all, create a new Unity project and copy the "RosSharp" folder from the ROS# GitHub repository (ros-sharp/Unity3D/Assets/RosSharp) into your Unity project's assets folder. To subscribe to and render point clouds, we are also going to make use of a GitHub repository called "unity_assets". This repository contains scripts that expand on ROS# and allows us to subscribe to and render a point cloud. So also add the PointCloudStreaming folder from this repository (unity_assets/PointCloudStreaming) to your Unity assets folder.

Make sure that Unity is using .NET Framework 4.x, since it is required by RosBridgeClient. To do this:

- In the Unity menu, go to `Edit > Project Settings > Player`.
- In the Inspector pane, look under `Other Settings > Configuration`.
- Set `Scripting Runtime Version*` to `.Net 4.x Equivalent`.

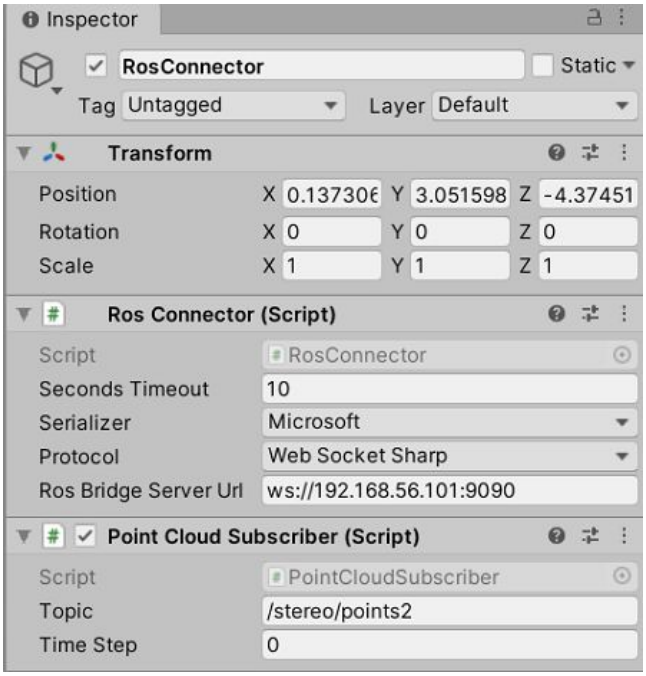
Now RosBridgeClient and UrdfImporter are included in your Unity project. Once the plugins have been loaded, the following new menu items will show up:

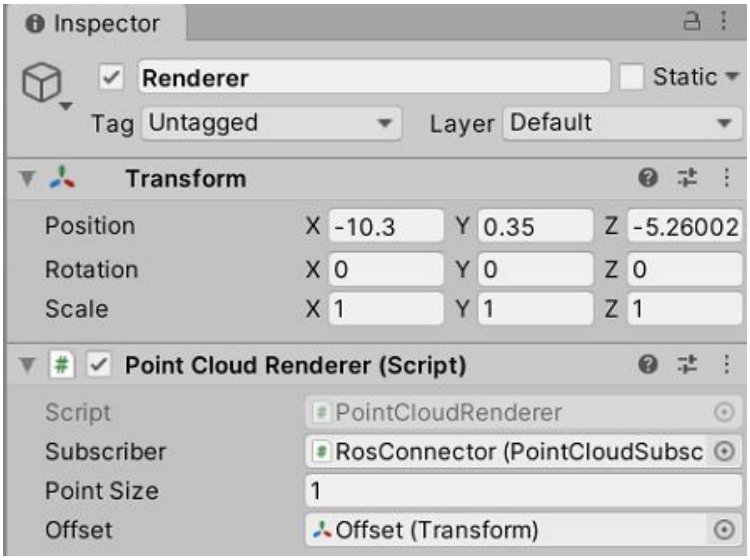


Next up, we need to set up our Unity objects. To see the point cloud in Unity, we will need the following objects:

- RosConnector:
  This is an empty object, inside this object we will add 2 scripts:
  - Ros Connector:
    This is a ROS# script that will connect the Unity project to the ROSbridge. Make sure that the server URL is inputted correctly.
  - Point Cloud Subscriber:
    This script subscribes our Unity project to a point cloud topic. Make sure the topic is the correct topic you want to subscribe to.

- Renderer:
  This is also an empty object with just 1 script:
  - Point Cloud Renderer:
    This script renders the point cloud once we receive data from the ROSbridge. It will spawn the point cloud in the position of the last object. This script is also linked to the PointCloudSubscriber inside of the ROS Connector.
- Offset:
  This item is linked to the renderer and will act as the base for the point cloud once it's rendered.

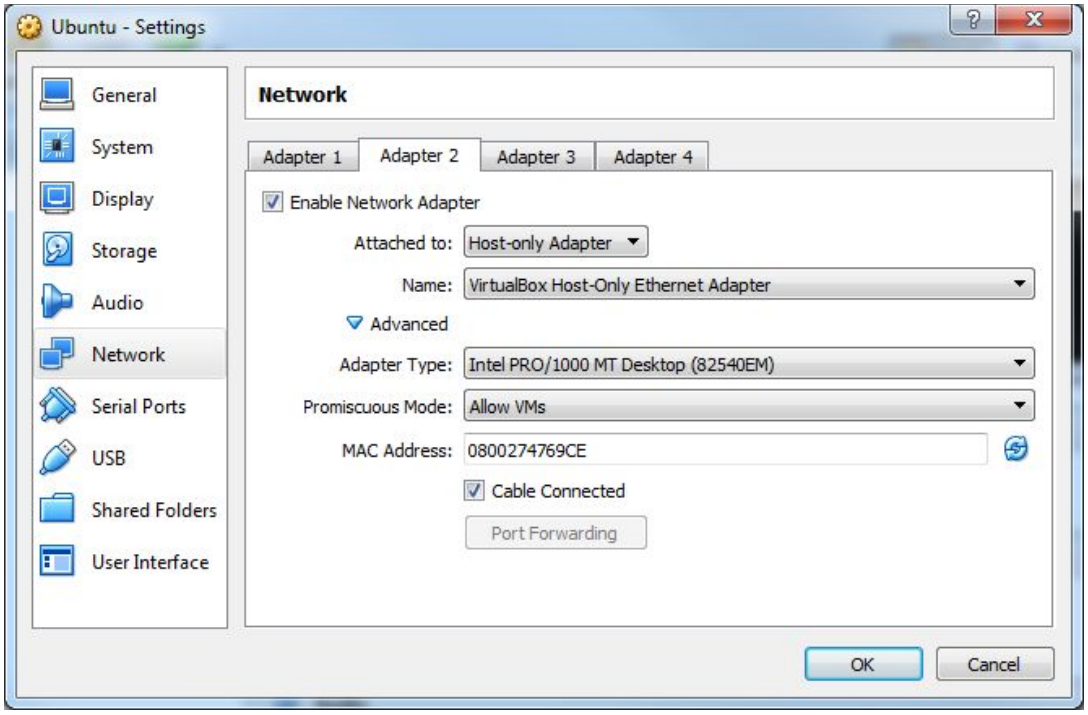




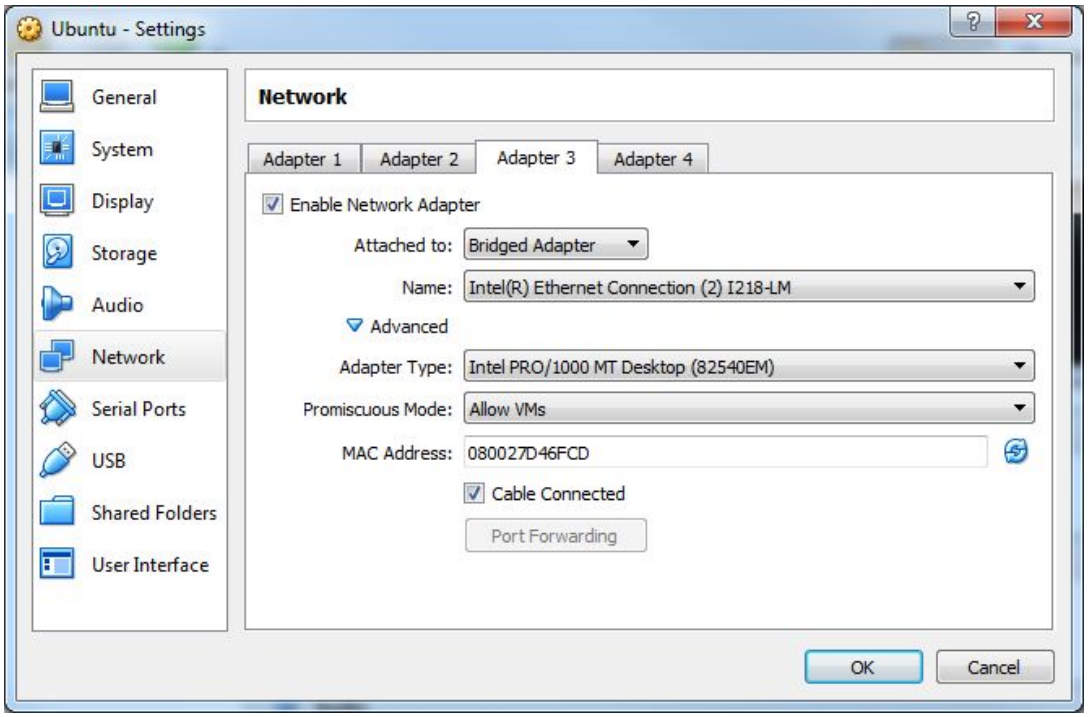
And finally make sure that you have added the "Enable Open GL" script to the camera.

For the next step we will take a look at our Ubuntu machine. In our case we use virtual machines. If you are not using a virtual machine but an actual Ubuntu machine you can skip this step. In Oracle VirtualBox, make sure the Ubuntu VM is powered off. Then open the settings for the Ubuntu VM. In the Network tab, add two new network adapters:

- Host-only adapter:



- Network bridge:



These settings are needed so that the RosBridgeClient running in Windows, and the ROSBridge Server running on Ubuntu can communicate. In Ubuntu type: `$ ifconfig` to check the network configuration and to verify your Ethernet connection to the Windows OS. The IP address (`enp0s8`) of the Ubuntu system will be used by `rosbridge_suite` and RosBridgeClient.
Now we need to set up the connection from the ROS side. Luckily, we don't need to write our own publisher and subscribers, instead we will make use of the ROSBridge. This is a part of ROS and is an easy way of establishing a connection between ROS and Unity. To install ROSBridge the user simply has to enter the following command into the terminal:

```
$ sudo apt-get install ros-<rosdistro>-rosbridge-suite
```

<rosdistro> is the distribution of ROS the user is using, for our project it is "melodic".
To now run the ROSBridge the user needs to first source the setup of their ROS directory:

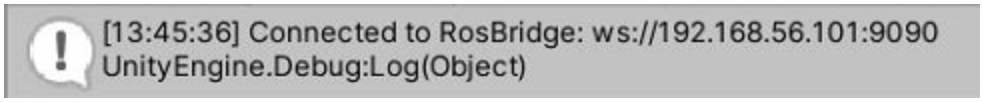
```
$ source /opt/ros/<rosdistro>/setup.bash
```

Now all the user has to do is run the following command:

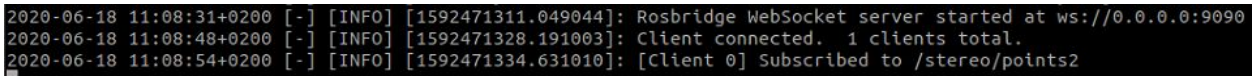
```
$ roslaunch rosbridge_server rosbridge_websocket.launch
```

And that is it, ROSBridge will now launch and create a websocket on port 9090 by default. Now it's time to play the Unity project, if everything went well you should see the following messages:

- In Unity:



- In ROS:



  It is important that it notifies you of the subscription as well as the connection. If one of these two messages is missing something has gone wrong.

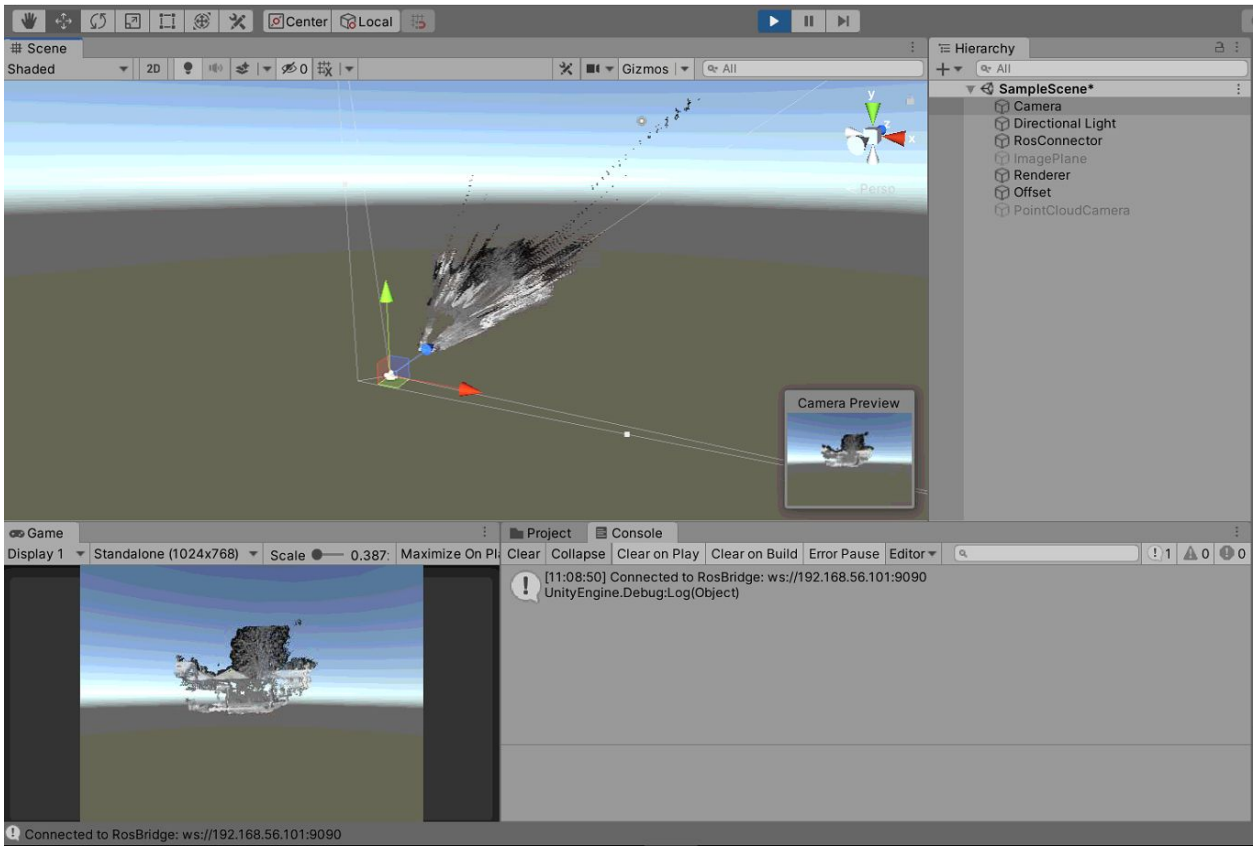
Now that the connection is established, it is time to start streaming data to Unity. In a separate terminal, navigate to your custom made package and run the command to generate a point cloud from the ROSbag:

```
$ cd ~/catkin_ws/src/pointcloud_visualizer
$ roslaunch pointcloud_visualizer stereo_image_proc.launch
```

While the stereo_image_proc.launch is getting launched, we can also start playing the ROSbag by playing it from the "launch" directory:

```
$ cd ..
$ rosbag play [name of rosbag]
```

Now, when you open up the Unity project you should be able to see the point cloud, which looks like this:



Sources:
https://github.com/inmo-jang/unity_assets/tree/master/PointCloudStreaming
https://github.com/siemens/ros-sharp
https://github.com/siemens/ros-sharp/wiki/User_Inst_Unity3DOnWindows
https://github.com/siemens/ros-sharp/wiki/User_Inst_UbuntuOnOracleVM
http://wiki.ros.org/rosbridge_suite/Tutorials/RunningRosbridge