LiDAR Scanner GUI Design

Yuan Han

Supervisor: Maciej Trzeciak

GitHub: https://github.com/Yamada-Yoshifumi/scanner-gui

September 2022

Contents

Chapter 1

Introduction

The objective of my summer internship is to produce a user-friendly, light-weight, modern and touchscreen-based graphical user interface (GUI) for bridging with a ROS/ROS2[1] LiDAR[2] scanner back-end. The back-end will have the following features available to clients: point cloud[3] reconstruction, texture mapping & reconstruction, and video streaming. As a result, the front-end user interface is designated to toggle, visualise, and tune parameters of those back-end processes. In this report the workflow of coding and several design ideas will be explained, with reference to a list of tools, packages and design created by others.

The main body of this report will focus on the ROS1 version, as the code structure remains mostly unchanged switching to ROS2. However, necessary alterations made for the ROS2 version will be explained in the third chapter.

[1] ROS: Robot Operating System

[2] LiDAR is a method for determining ranges (variable distance) by targeting an object or a surface with a laser and measuring the time for the reflected light to return to the receiver

[3] A point cloud is a set of data points in space

Chapter 2

The GUI

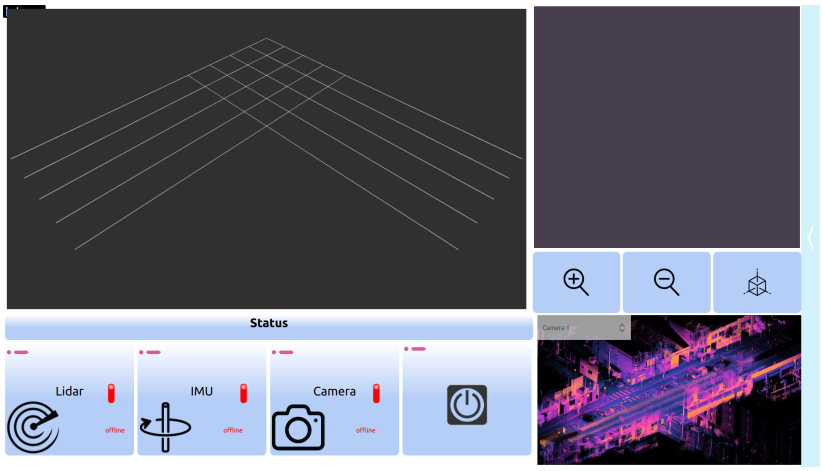

Figure 2.1: GUI, daylight mode.

Qt[1] is a good candidate for this job, thanks to its object-oriented code structure, compatibility with different programming languages, compatibility with ROS, and convenient integrated tools such as the animation classes and the QThread class. My experience with ROS development in C++ was another decisive factor when this decision on development platform was made. As a result the overall code structure became: C++ Qt classes with ROS methods and objects as class properties.

Figure 2.1 is the final design of the GUI. It comprises of: 1. an RViz render panel to display real-time point cloud and reconstruction; 2. a touch pad to control the view angle and the view distance; 3. a hardware status panel to indicate hardware package status; 4. a video streaming window; 5. a settings panel hiding near the right edge. The status panel, the streaming window, and the settings panel are made with QML and built-in JavaScript, as the parts written with QML are easier to animate, and they work more seamlessly with touch screen. For everything to be correctly positioned and displayed, there also has to be a MainWindow class to wrap everything together.

2.1 Main Window

The MainWindow class is defined in mainwindow.h. It initialises QML containers for main QML code and settings QML code, a MyViz instance (an RViz group), and miscellaneous objects such as timers. Besides its responsibility of holding all the QWidgets and displaying all in a window, its main functionality is to initialise QObjects and create pointers to QObjects (Buttons, Images, Colours, etc.) in both QML and C++

[1] Qt is a cross-platform software development tool

code, and connect the signals of these QObjects to C++ methods. Those C++ methods are categorised as Q_SLOTS properties which respond to signals emitted by the QObjects. For example, the power button is connected to a systemPowerToggle() method in ROSHandler class via two connections:

```
//Set a pointer to the power button
qmlView = new QQuickView();
qmlView->setSource(QUrl(QStringLiteral("qrc:/qml/App.qml")));
QObject *item = qmlView->rootObject();
power_button = item->findChild<QObject*>("power_button");
//The SLOT powerClickedEmit() method emits the powerButtonPressed signal used in the second
    connection.
connect( power_button, SIGNAL( powerSignal(QString) ), this, SLOT( powerClickedEmit() ));
connect( mainwindow, &MainWindow:: powerButtonPressed, roshandler, &ROSHandler::
    systemPowerToggle, Qt:: QueuedConnection);
```

, in order to start the Velodyne node upon a button click. Some similar implementations require two connections, but some requires one only, as only the signals from a C++ QObject class can be connected to class methods, so an intermediate remapping is required for those signals coming from QML.

Another feature of MainWindow is that it listens to the camera, Velodyne and Imu topics via ROS subscription, in order to determine if the packages are online. Therefore, the MainWindow object is able to update the indicators in the hardware status panel in the bottom left. The algorithm is: the QTimer objects are started for each of the hardware packages, and reset the corresponding hardware status to "inactive" every second. The subscription callback functions are designed to restart the timers, so that the timers don't reach the callback stage, and furthermore they do not get to reset the status variables. See the code block below for more details.

```
//subscription callback
void MainWindow::updateVelodyneStatus(const sensor_msgs::PointCloud2ConstPtr &msg){
    velodyne_timer->start(1000);
    velodyne_status = 1;
    paintStatus();
}
//QTimer callback
void MainWindow::resetVelodyneStatus(){
    velodyne_status = 0;
    if(power_button_bg != nullptr)
        paintStatus();
}
```

2.2 Qt RViz Plugin

The highest priority during development of this GUI was to embed the RViz[2] window into the Qt GUI, for visualising real-time point cloud and reconstruction. Due to limited compatibility of RViz libraries[3] on Qt platform, Qt 5.15 LTS was the only option for this GUI to be developed on.

[2] RViz: A 3D visualisation tool for ROS
[3] Librviz

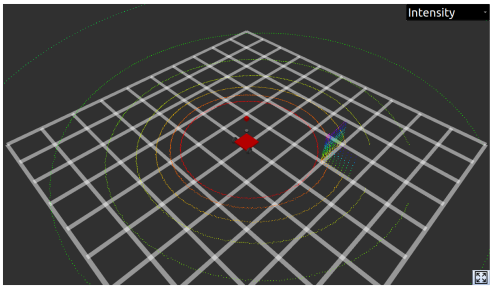

Figure 2.2: Embeded RViz

Figure 2.2 demonstrates how one would use the RViz panel. The panel looks, and works the same as the RViz GUI client. The real-time point cloud and the robot model are displayed. The drop-down box at the top right corner sets the colour pattern of the point cloud, which currently supports AxisColor, Intensity, FlatColor, and Uncertainty.

The RViz plugin utilises the original RViz libraries, including "visualization_manager.h", "render_panel.h", and "display.h", from https://github.com/ros-visualization/rviz. The RenderPanel class is used to initialise a rendering panel, on which different kinds of Display objects, e.g. point cloud, can be drawn. The VisualizationManager class sets the properties of the view camera, such as yaw, pitch and view distance. The objects above are included in the MyViz class, defined in myviz.h.

```
render_panel_ = new rviz::RenderPanel(this);
render_panel_->installEventFilter(this);
manager_ = new rviz::VisualizationManager( render_panel_ );
render_panel_->initialize( manager_->getSceneManager(), manager_ );
manager_->initialize();
manager_->startUpdate();
//A fixed frame at base_footprint makes the view move along with the robot
manager_->setFixedFrame("base_footprint");
grid_ = manager_->createDisplay( "rviz/Grid", "adjustable grid", true );
pointcloud_ = manager_->createDisplay("rviz/PointCloud2","PointCloudAir", true);
tf_ = manager_->createDisplay("rviz/RobotModel","lidar tf", true);
```

2.3 Touch Pad

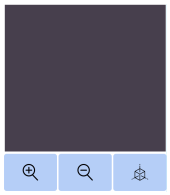

Figure 2.3: Touch Pad

The touch pad[4] consists of three QPushButton instances and a TouchPad instance. The grey-looking rectangle in Figure 2.3 is the touch pad, which reads up to two-finger QTouchEvents and correspondingly changes the camera pitch, yaw, view distance or view position of the RViz window by feeding the generated event to the RenderPanel object in RViz. Classification of gesture is achieved by detecting the relative motion of two finger tips. If they are moving towards each other, it is a pinching gesture, so a QWheelEvent which emulates mouse wheel scrolling will be fed to the render panel.

[4]The touch pad can be overlapped by a debug message window which will be explained later in this chapter

The render panel has an event filter installed, which means it can filter the received events, and update the view properties upon receipt of certain events. This TouchPad instance is necessary because the RViz classes are based on a 3D graphics engine called Ogre[5], which does not support multi-touch events. As is mentioned above, the GUI will be completely touchscreen-based. Therefore this TouchPad as a workaround is needed. An overlaying transparent touch widget appears to be another possible solution. However, Qt requires both the underlying and the overlaying widgets to be QGraphicsItems[6], which is impossible because the RViz window cannot be converted so that it inherits from such a class.

```
1  //example in event feeding in TouchPad. _trunc_pos_1 and _int_win_pos are points of the
       touch event, converted from QPoints on the touchpad, to QPoints on the render panel
2  if(event->type() == QEvent::TouchUpdate){
3  QMouseEvent event(QEvent::MouseMove, _trunc_pos_1, _trunc_pos_1, _int_win_pos, Qt::NoButton, Qt::
       LeftButton, Qt::NoModifier);
4  this->parentWidget()->eventFilter(this->parentWidget(), &event);
5  }
6
7  //example in event filtering in MyViz for RenderPanel, increment yaw
8  QMouseEvent* pMouseEvent = dynamic_cast<QMouseEvent*>(p_event);
9  if((pMouseEvent->globalPos().x() - previous_touchp.x()) > 5){
10     current_yaw += 0.05;
11     manager_->getViewManager()->getCurrent()->subProp("Yaw")->setValue( current_yaw );
12     previous_touchp = pMouseEvent->globalPos();
13
14 }
```

The three QPushButtons are of the similar mechanism. The zoom in/out buttons change the view distance by calling methods in VisualizationManager, and the reset-view button resets view distance, yaw, and pitch.

One possible alternative to the current algorithm is to use existing Qt gesture libraries[7]. They provide better precision at determining the current gesture. However, there is too little documentation on this feature. In addition to that, QTouchEvent is still necessary for one-finger touch, and it is going to be messy if the event handler needs to handle both QGestureEvents and QTouchEvents, as their detection will overlap. Nevertheless, this Qt gesture solution should be considered if we are looking into improving the accuracy of gesture classification.

2.4 Status Panel

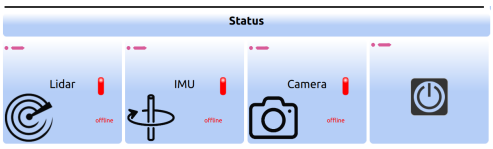

Figure 2.4: Status Panel

The status panel[8] is written with QML and JavaScript, because in QML, the Layout management is easier, and it is easier to incorporate the Icons which will be animated. Each section has an indicator drawn by JavaScript Canvas, a section Icon, and text to indicate the current status of the corresponding hardware. The paintStatus() method in MainWindow class repaints and changes the colour of those objects.

[5] Ogre
[6] QGraphicsItem
[7] QGestureEvent
[8] Art attribution: Camera icon created by Good Ware; Axis icon created by Freepik; Radar icon created by Lorc; Power icon

2.5 Video Streaming Window

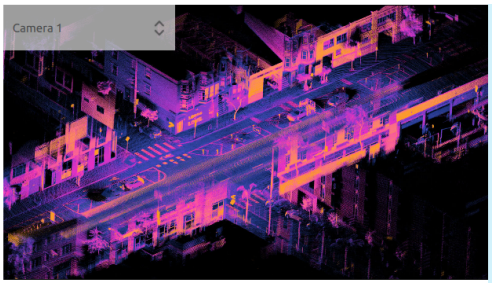

Figure 2.5: Video Streaming Window

The video streaming window is also implemented with QML, because existing QML packages can be used for this feature, such as QQuickImageProvider. One OpenCVImageProvider class is defined in opencvimageprovider.h to act as an image provider to the Image object in QML; One VideoStreamer class is defined in videostreamer.h to convert sensor_msgs::Image frames to OpenCV frames utilising the cv_bridge package[9], and feed them to the image provider via a signal-slot connection explained above in section 2.1. The logistics of this video stream is as follows:

```
1  //In a VideoStreamer Object
2  //Image subscription callback function
3  void VideoStreamer::convertROSImage(const sensor_msgs::ImageConstPtr &msg){
4      init = true;
5      current_frame_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::RGB8);
6  }
7  //newImage signal emitted if a frame is present
8  void VideoStreamer::streamVideo()
9  {
10     if(current_frame_ptr != nullptr && init){
11         QImage img = QImage(current_frame_ptr->image.data,current_frame_ptr->image.cols,
               current_frame_ptr->image.rows,QImage::Format_RGB888);
12         emit newImage(img);
13     }
14 }
15 //Connection in mainwindow.cpp
16 connect(videoStreamer,&VideoStreamer::newImage,liveimageprovider,&OpencvImageProvider::
       updateImage);
17 //In OpencvImageProvider Object
18 //imageChanged emitted if the new image is different to the previous frame
19 void OpencvImageProvider::updateImage(const QImage &image)
20 {
21     if(!image.isNull() && this->image != image) {
22         this->image = image;
23         emit imageChanged();
24 }
25 //in mainwindow.cpp, imageReload called upon receipt of imageChanged signal
26 connect(liveimageprovider, SIGNAL(imageChanged()), this, SLOT(imageReload()));
27 void MainWindow::imageReload(){
28     if(videoStreamer->init && videoStreamer->current_frame_ptr != nullptr)
29     {
30         if(counter){
31             opencv_image->setProperty("source", "image://live/image?id=1");
32         }
33         else
34         {
35             opencv_image->setProperty("source", "image://live/image?id=0");
```

[9] cv_bridge

```
36          }
37          counter = !counter;
38      }
39  }
```

The OpencvImageProvider class inherits from the QQuickImageProvider class, which makes it generate alternating paths, "image://live/image?id=1" and "image://live/image?id=0", for the current frame. In other words, the current frame is recognised as a resource file just like every other image, icon or font file, with the alternating paths generated as the link to it. As it is shown in lines 27 to 39, the video streamer works in a way such that the current source of the Image object in QML is always being changed, corresponding to the ever-changing live image path generated by the OpencvImageProvider object.

The combobox at the top left corner is able to switch between two video sources. Its current selection is stored in an sqlite3 database (will be explained later in the Settings Panel section) and read by a C++ thread to change the ROS Image topic subscription, instead of emitting a signal whenever the current selection is changed. This communication method is used in other parts of the program as well, when the offline storage of the variable is needed, and when response speed is not significantly important.

2.6 Settings Panel

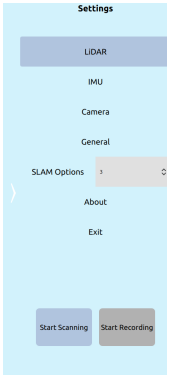

Figure 2.6: Settings Panel

The settings panel is again, completely written in QML and JavaScript. This decision was made upon the good MVC[10] libraries QML has. QtQuick.Controls and Qt.labs.qmlmodels provide ListModel, Delegate and ListView, for smooth swiping gesture response and better code structure. QtQuick.LocalStorage provides a JavasScript library for easy read&write access to a local sqlite3 database, in which user-preferred settings can be stored.

Using the StackView class in QML, the settings panel can push in or pop up pages with sliding animation upon click event on any entry or on the return button. All of the settings will be accessed by other QML pages and the C++ objects, in order for the change in settings to take effect. For example, the daylight mode toggle is under the General entry, and if it is switched off, the instance will emit a signal to other QML pages, reminding them of changing the colour theme; In the mean time, a C++ thread which spins once every second will read the change in the sqlite3 database, and change the colour theme of the C++ QObjects. See Appendix 1 for a complete list of available settings.

[10] MVC: Model, View, Controller

2.7 Debug Window

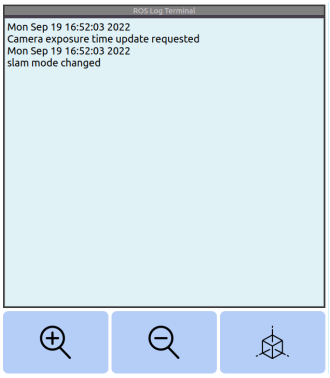

Figure 2.7: Debug Window

The presence of a debug window in the code is for the sake of developers. In the ROSHandler class, a signal will be emitted back to a C++ thread once a ROS service client is executed successfully. The message that is carried by the signal is then logged into the debug window in the C++ thread. This process facilitates debugging of the entire system, as it tells the developer if the bug lives in the front-end or the back-end.

2.8 Miscellaneous

2.8.1 QML ROS Plugin

The signal-slot connections and database read&write may seem to be complicated communication methods between QML and C++. Indeed, there is a powerful open-source QML ROS plugin available[11]. Despite its simplicity and power demonstrated in the release note[12], its dependencies are quite tricky to install, at least it is so on my personal laptop running Ubuntu 20.04 with ROS Noetic. The objective of this project is to build a client-oriented product, so it is my personal view that with rather complicated installation requirements, the QML ROS plugin is not very promising. However, if one figures out how to configure the environment easily, a partially-QML program with the ROS plugin approach can be a strong candidate.

As it is mentioned before, the usage of QML in this "C++ based" program is for better animation, graphics and touchscreen interactions. If these merits can be compromised, an 100% C++ program is going to make the code significantly more readable.

2.8.2 Threading

Usage of threading has been mentioned a few times so far. There are currently two threads in use: one is the QThread object in main.cpp, the other is a C++ thread object in the hardware simulation package velodyne_description. The first QThread exists because in the main function in main.cpp, app.exec() has to be called to start the QApplication, and only continues to the next line if the QApplication instance has been closed. Therefore, a thread is started before app.exec() is called. Putting this QThread here in main.cpp is desirable, as otherwise the QThread will need to be in mainwindow.cpp, which is already rather complicated in code structure. The second C++ thread is for "roslaunching" a launch file from within a C++ ROS node. Because the new nodes will be running for a long time, execution of the callback can not be awaited. Thus this thread is necessary as well.

[11] From StefanFabian
[12] https://discourse.ros.org/t/qml-ros-plugin-release/22096

The QThread called power_thread in main.cpp is in charge of calling methods in the ROSHandler object, and reading the database once every second to make changes accordingly. The ROSHandler methods invoked here send out ROS service request to services in the back-end ROS packages, to either start a package or update the parameters of a package. See an example of updating the back-end camera exposure-time below:

```
//In QThread, reading the corresponding row in the database
if((sqlite3_step(stmt)) == SQLITE_ROW) {
    if(sqlite3_column_int(stmt, 1) != database_camera_exposure_t){
        database_camera_exposure_t = sqlite3_column_int(stmt, 1);
        //Call the method if change was made
        roshandler->cameraExposureUpdate(database_camera_exposure_t);
    }
}
//In ROSHandler
void ROSHandler::cameraExposureUpdate(int database_camera_exposure_t){
    cameraExposureUpdateSrv.request.command = database_camera_exposure_t;
    if(cameraExposureUpdateClient.call(cameraExposureUpdateSrv)){
        //The message emitted with this signal will be logged into the debug window in 2.7
        emit cameraExposureUpdatedSignal("Camera exposure time update requested");
    }
}
```

If the service request is for starting a hardware package, this brings us to the second thread. The second thread is called velodyneOnThread. It makes use of the ROSLaunchManager class defined in roslaunchmanager.h[13], in order to start a launch file from with a ROS node. The launch file starts gazebo_ros simulation with a custom world, robot_state_publisher[14], joint_state_publisher[15], and a Velodyne LiDAR model in it.

2.8.3 Simulation

The gazebo_ros simulation is provided from https://bitbucket.org/DataspeedInc/velodyne_simulator/src/master/. Minor adaptions has been made to add a camera and an IMU on board, and to fix the Velodyne LiDAR onto a chassis. A velodyne_control package was written to send velocity command to the LiDAR chassis.

[13] Adapted from Tannz0rz's code
[14] Robot State Publisher
[15] Joint State Publisher

Chapter 3

ROS2 Humble Alterations

3.1 ROS2 Subscriptions and Service Clients

Migrating to ROS2 Humble apparently means that the previous ROS1 subscriptions and service clients need to be rewritten. Service clients in ROS2[1] do not differ much from their ROS1 equivalent, but subscriptions[2] have been changed a lot. In short, the changes are:

- Quality of Settings(QoS)[3] needs to be set for a subscription to be universally compatible with many publishers
- Subclassing the ROS Node class is encouraged for subscription and callbacks definition

For easy subclassing, the ROS Node has been moved to main.cpp, along with the subscriptions to hardware package topics. In addition, the RViz plugin "myviz" is the main window object now (will be explained in the next section), and it's pointer is declared as a global variable, so that the callback functions in ROS Node "ROSMessageDetector" can easily change the colours and animations in the status panel in myviz.

A specific change related to camera subscription is that we have included cv_bridge into our package, and have renamed it to cv_bridge_ros2. Cv_bridge was briefly mentioned in section 2.5 Video Streaming Window, and it is used to convert ROS Image frames into OpenCV Image frames. The reason for this renaming is that ROS Noetic already has cv_bridge installed on my machine, and possibly will does so on clients' machines as well. This ROS Noetic cv_bridge package was always mistaken as the new ROS Humble cv_bridge when compiling the ROS 2 version of the GUI. Therefore, I have included the new package with a new name for clarity.

3.2 Qt RViz Plugin

As the most important part of the GUI, unfortunately, the RViz library has undergone many updates, which means this part of the ROS 1 code needs to be heavily modified.

One of the most significant updates is that the new RViz library, rviz_common[4], has almost every class as an abstract class[5]. More importantly, in order to contain and display the RenderPanel and the Display widgets, the parent class "MyViz" has to inherit from rviz_common::WindowManagerInterface and QMainWindow now. Those two facts force the MyViz class to also be an abstract class, and the MyViz class has to act as the main window.

This leads to some change in the code. The MainWindow class is no longer necessary, because the MyViz class is QMainWindow now. All of the necessary connections, instances, and methods are transferred from

[1] ://github.com/ros2/examples/tree/humble/rclcpp/servicesROS2 service examples
[2] ROS2 topic examples
[3] rclcpp::QoS
[4] rviz_common
[5] An abstract class is a class with at least one pure virtual member function

the previous MainWindow class to the MyViz class. Furthermore, because MyViz is now an abstract class, one can no longer initialise an instance from it. Instead, a shared pointer[6] is created to reference to the abstract class, and to access all the member functions and properties in it.

There are alternative ways to adapt to this change. Firstly, one can subclass the rviz_common:: WindowManagerInterface class, and include implementations of all the parent's virtual functions to make the new class instantiatable. However, the MyViz class still need to be the main window, because an inheritance from QMainWindow is still required. Secondly, one can stick to the ROS 1 RViz libraries, and use ros1_bridge to communicate with back-end ROS 2 packages. The final decision was to use a shared pointer, because it takes the least effort, gives clean code, and doesn't add too much new code on the base of the ROS 1 version.

The new library has also made the RViz RenderPanel window always display on top of other widgets because the window is an abstract class and hence this breaks the parent-child management system in Qt. In this scenario, neither the colour pattern combobox nor the full-screen button can be overlaying the render panel anymore. As a result, both objects have been removed, but the colour pattern can still be set in the settings panel.

The other change made due to this abstract class problem was in class TouchPad. The new rviz_common:: RenderPanel no longer processes QEvents, so TouchPad can no longer feed QMouseEvents or QWheelEvents to RenderPanel. Instead, because the MyViz class is the main window now, it can be referenced by:

```
1
2 auto mainWin = qobject_cast<MyViz*>(window());
```

Pointer mainWin therefore references to MyViz. The properties of the ViewManager in MyViz, including yaw, pitch, view distance, etc. can be directly modified from the TouchPad event handler, for example the view distance:

```
1
2 mainWin->current_f_distance += pixelDelta.y()/10;
3 mainWin->manager_->getViewManager()->getCurrent()->subProp("Distance")->setValue( mainWin->
      current_f_distance );
```

Last but not least, for rviz_common in ROS2, an rviz_common::ros_integration:: RosNodeAbstraction[7] shared pointer is required during the creation of the rviz_common:VisualizationManager object. It provides a shared clock for VisualizationManager, and eventually is passed to an rviz_common::transformation:: FrameTransformer[8], acting effectively as a common ROS node which listens to transformations. In rviz in ROS1, the nodes have always been initialised by the classes themselves. Therefore there was no need passing a Node pointer to the old VisualizationManager.

3.3 Lifecycle Management

Lifecycle Node management[9] is a way to manage the status of nodes in ROS 2. It is not officially added into the program yet because the front-end and the back-end have not been interfaced, but it is certainly going to happen in the future. A node management client was added into the ROSHandler class, and it controls the deactivation, the cleanup, and the shutdown of other nodes via ROS 2 service requests. It will be a more elegant way to shutdown nodes than what we have in the ROS1 code: "rosnode kill $< nodename >$" is executed via the "system()" command in C++ to shut down nodes one by one.

[6] Shared pointer, a type of smart pointer
[7] https://github.com/ros2/rviz/blob/humble/rviz_common/include/rviz_common/ros_integration/ro_node_abstraction.hpp
[8] https://github.com/ros2/rviz/blob/humble/rviz_common/include/rviz_common/transformation/frame_transformer.hpp
[9] Lifecycle Management

Chapter 4

Conclusion

After 9 weeks of development, both the ROS 1 and ROS 2 versions of the GUI were in working state, with the help from my supervisor Maciej Trzeciak, open-source packages and the developers behind them (See Appendix B). The GUI fulfilled its original design goals of being easy to use, light-weight and modern, and alternative approaches to problems were raised and discussed during the development process. Many of those alternatives remain worthy of consideration when it comes to improving the GUI. During integration with real-world back-end packages, the GUI is modular enough that only the service names and messages in ROSHandler defined in roshandler.h (briefly mentioned in section 2.8.2 Threading) will need to be changed. Nevertheless, in order for the GUI to fully function, advanced features, including settings entries and Debug Window, will need to be updated as well. Future development may include automatically updated service clients and debug window messages from the added entries of the settings panel, along with other possible improvements that have been discussed: adding QGestureEvent to touch pad, adding QML ROS Plugin, and lifecycle node management.

Appendix A

List of Settings

- LiDAR
  - Reconstruction
  - Default Colour
- Camera
  - Exposure Time(ms)
- General
  - Daylight Mode
  - Debug Mode
- SLAM Options

Appendix B

List of Open-Source Packages Used or Adapted

- rviz library
- rviz_common library
- Qt 5.15.2 packages
- qml-modules
- ros-noetic-desktop-full
- ROS2 Humble from source
- librviz
- OpenCV 4
- cv_bridge
- vision_opencv
- severin-lemaignan/ros-qml-plugin
- sgully/qml-examples
- velodyne_simulation
- Tiryoh/docker-ros-desktop-vnc
- Tannz0rz's ROSLaunchManager