NTNU – Trondheim
Norwegian University of
Science and Technology

Visualization System for Robot Cells

Rune Bilit

Master of Science in Engineering Cybernetics
Submission date: June 2012
Supervisor: Geir Mathisen, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

NTNU
Norges teknisk-naturvitenskapelige universitet

Fakultet for informasjonsteknologi, matematikk og elektroteknikk
Institutt for teknisk kybernetikk

HOVEDOPPGAVE/MASTER THESIS

Kandidatens navn: **Rune Bilit**

Fag: **Teknisk kybernetikk/Engineering Cybernetics**

Oppgavens tittel (norsk): **Visualiseringssystem for Robotceller**

Oppgavens tittel (engelsk): **Visualization System for Robot Cells**

This thesis shall propose a 3D system for visualization of the activity in a robot cell. The proposed system should be a tool for real-time visualization of the robot activity, for instance to use in remote monitoring. The tool should also be able to play a visualization of the process in retrospect, both in true and slow motion, e. g. for documentation or fault diagnosis.

The proposed tool should be based on displaying models of known objects, e. g. the robot itself and known objects in the cell, but also handle unknown objects detected by the used sensors. The tool should have the ability to represent important process variables (e. g. forces and speeds) in real-time, and show where in the 3D scene the variables are active.

The main focus of this thesis should be the needed software to implement the proposed tool.

The following tasks shall be addressed:

1. Give a literature survey of exising open source software relevant for this thesis.
2. Propose a system for visualization of the activity in a robot cell.
3. As fare as time allows, implement the system proposed in paragraph 2.

Oppgaven gitt: 9th January, 2012
Besvarelsen leveres: 4th June, 2012

Utført ved Institutt for Teknisk kybernetikk

Faglig veileder: Johannes Schrimpf

Trondheim, den 9.01.2012

Geir Mathisen
Faglærer

Abstract

Visualization is a technique for creating a graphical representation of a reality, with possible augmentations. It can be used in a wide variety of applications in the area of robotics. An example usage is for teleoperation, where a human operator can control a robot from a different location by having the robotic process visualized in real-time. Furthermore, visualization software can also be used for doing detailed analysis of a process, for instance to debug a system that is under development.

This thesis proposes and implements a system that visualizes robot cells in 3D. The system presented here uses RViz as the actual visualization program, and Robot Operating System (ROS) to connect the different parts of the system together. In all, the system is organized in a way that makes it easy to combine several different robots and other devices in one view. The main part of the system presented in this thesis is a Graphical User Interface (GUI), which is used to control the actual visualization, as well as the rest of the system. This GUI provides functionality to record and play back measurement data from the different devices and sensors, making it possible for a human operator to pause, resume and step through the data one value at a time. To be able to use this general system on a real robotic process, it has to be customized by specifying the available devices, and information about these, in a configuration file.

The developed system was tested on the Ekornes sewing cell at SINTEF Raufoss Manufacturing. The sewing cell consists of a sewing machine and two industrial robots, that are used for sewing together leather covers. Based on this testing it was seen that the program is a powerful tool for analyzing robotic processes, because of the possibility to record and replay data, as well as providing the operator with a very informative visualization view.

Sammendrag

Visualisering er en teknikk for å skape en grafisk fremstilling av en virkelighet, med mulige endringer. Det kan brukes i en rekke forskjellige applikasjoner innen robotikk. Et bruksområde er for teleoperasjon, hvor en menneskelig operatør kan styre en robot fra et annet sted, ved å se en visualisering av prosessen. Videre kan visualiseringsprogramvare også brukes for å gjøre detaljerte analyser av en prosess, for eksempel for å feilsøke et system som er under utvikling.

Denne avhandlingen foreslår og implementerer et system som visualiserer robotceller i 3D. Systemet som presenteres her, bruker RViz som selve visualiseringsprogrammet, og Robot Operating System (ROS) for å koble de ulike delene av systemet sammen. I alt er systemet organisert på en måte som gjør det enkelt å kombinere flere ulike roboter og andre enheter i ett og samme bilde. Hoveddelen av systemet som blir presentert i denne avhandlingen er et grafisk brukergrensesnitt, som brukes til å styre selve visualiseringen, samt resten av systemet. Dette brukergrensesnittet gir funksjonalitet for å ta opp og spille av måledata fra de ulike enhetene og sensorene, noe som gjør det mulig å stanse, fortsette og gå gjennom målingene én etter én. For å kunne bruke dette generelle systemet på en ekte robotprosess, må det tilpasses ved å angi de tilgjengelige enhetene, samt informasjon om disse, i en konfigurasjonsfil.

Det utviklede systemet ble testet på Ekornes-sycellen ved SINTEF Raufoss Manufacturing. Sycellen består av en symaskin og to industriroboter, som brukes for å sy sammen lenestol- og sofaputer. Basert på denne testingen ble det sett at det resulterende visualiseringsprogrammet er et kraftig verktøy for å analysere robotprosesser, på grunn av muligheten for å ta opp og spille av data, samt gi operatøren et meget informativt visualiseringsbilde.

Preface

This master thesis is written at the Department of Engineering Cybernetics at NTNU during the spring semester of 2012. First, I would like to thank my supervisor Geir Mathisen for guidance and constructive feedback during this work. I would also like to thank my co-supervisor Johannes Schrimpf for giving me great help throughout the entire thesis. In addition, I would like to thank SINTEF Raufoss Manufacturing for providing me with the opportunity to do some real testing on the Ekornes sewing cell.

Trondheim, June 2012
Rune Bilit

Contents

Chapter 1

Introduction

1.1 Motivation

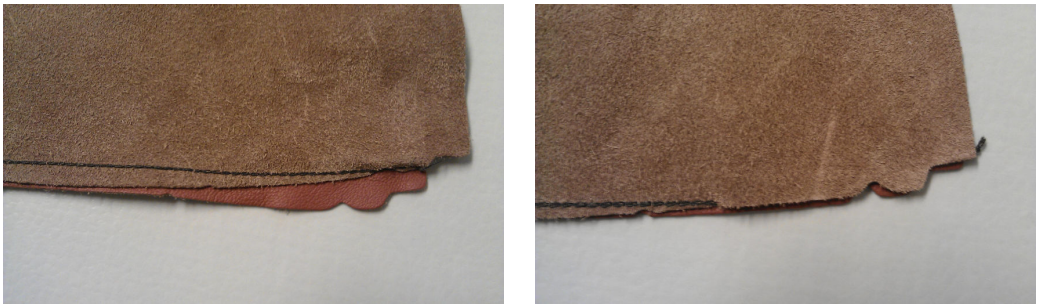
When developing new hardware and software applications, it is important to have good debugging tools. In the area of robotics, one approach for debugging is to use visualization software. Originally, data from different sensors are simply represented as numbers, and in many cases plotted as a graph or similarly. However, as the number of sensors increases, it gets harder to compare each of the sensor data readings separately and to create a complete overview of the system.

Through the use of 3D visualization it is possible to give a much more intuitive representation of the sensor data, and in addition, an easy way to combine data from different sensors. The sensor data can also be combined with other kinds of representations, such as a robot model. Furthermore, by using a vision sensor, a visualization tool makes it possible to provide a resulting view that consists of an augmented version of a real process, in which extra information is added. The resulting system can be a very powerful tool for analyzing details of the process.

3D visualization of a process can be done both online and offline. Here, *online* refers to running the visualization of a process in real-time, for instance to be used for teleoperation. Moreover, *offline* refers to visualizing a process by recreating it

based on recorded sensor data. Recreating a process offline makes it possible to use the 3D visualization to do detailed system analysis, for instance by playing back the data at a different speed, and by moving the camera view to look at the process from different angles and distances.

A specific application, in which a 3D visualization tool is useful, is the Ekornes sewing cell at SINTEF Raufoss Manufacturing. The sewing cell consists of a sewing machine and two industrial robots, where the robots are used to sew together leather covers for sofas and armchairs. One of the reasons for studying this process in detail, is because it is not completely consistent. That is, even though all control parameters are the same, the result from running two experiments might differ. Figure 1.1 illustrates this problem. Here, Figure 1.1(a) shows a badly sewn seam, while Figure 1.1(b) shows a better one. Even though the same parameters are used for sewing these covers together, the results are different. This is a problem that can be studied more detailed by having a 3D visualization tool that supports playing back data offline to look at the behavior of all devices and sensors at a specific time.



(a) A poor seam. (b) A better seam, though still not perfect.

Figure 1.1: Two leather covers sewed together. In both images, the same parameters are used, but the result still differs.

1.2 Background

Having a visualization tool for robotic applications requires a good software basis, for instance through the use of a robotic software framework. For the past years,

several different frameworks have been released. One of the currently most popular ones is Robot Operating System (ROS) [1]. ROS is by the developers described as a *meta-operating system* [2, 3], meaning it provides the typical services you would expect from an operating system, and in addition, provides tools and libraries for building, writing and running code across multiple computers. In addition to ROS, there are also several other robotic software frameworks available. A survey of these frameworks can be found in [4].

The reason for having a robotic software framework as basis for visualization software is because of all the different functionality it provides. This functionality includes hardware abstraction and ways to communicate between processes. By utilizing this functionality, the data to be visualized can easily be received from other processes, and different sensor data can easily be integrated with each other.

A commonly used 3D visualization tool is RViz [5], which is released as a visualization environment under ROS. One of the advantages of using this visualization tool is the tight integration with ROS. For instance, RViz is able to display several of the different standard data types available in ROS. Moreover, the displays can easily be combined into a complete view, consisting of several different devices and sensor data. A typical scenario is using ROS as a robotic platform for a system, and to test this system by visualizing it in RViz [6, 7].

1.3 Contribution

This thesis gives an overview of different available robotic software frameworks and 3D visualization software. Based on this overview, a 3D visualization software for robotic applications is developed, and described in detail.

The starting point for the development of a visualization tool is ROS and RViz. By utilizing all the available functionality in ROS, the resulting software is able to visualize a scene consisting of several robots and other devices. The visualization includes point clouds, static models of robots and other sensor data. *Point clouds* are generated from some sort of a depth sensor, such as the Kinect [8], and consist of several points in space. If supported by the depth sensor, each of the points have

a corresponding RGB value, which is shown in the visualization. Furthermore, the pose of the *robot models* are changed accordingly to the real robots, based on information about the current joint states. Finally, the *other sensor data* refers to specific sensor readings that are visualized by displaying the current data values and arrows to indicate the magnitude and direction of the corresponding data. These displays include the velocity and force sensor for each robot, as well as other device specific information.

The software developed for this thesis consists of a Graphical User Interface (GUI), where it is possible to select different devices that should appear in the visualization. All the available devices are specified in an configuration file, together with information about these devices. Furthermore, through the use of the GUI, it is possible to run the program in real-time or playback mode. Running the program in *real-time* mode gives a human operator the possibility to record data for the selected devices. In *playback* mode, on the other hand, it is possible to *play back* previously recorded data. The resulting visualization of the devices is the same in both program modes. However, by running in playback mode, the operator is able to do some more detailed analysis, by pausing and resuming the playback, or by stepping through the data one value at a time.

To use this general system for a specific application, it has to be slightly customized. Thus, the details around customizing this program for the Ekornes sewing cell at SINTEF Raufoss Manufacturing are described in this thesis. Through the use of this specific case, it can be seen that the system meets its specifications, and that it can easily be expanded to support other devices.

1.4 Outline

The rest of this thesis is organized by first presenting a literature study of currently available robotic software frameworks and visualization tools in Chapter 2. Then, in Chapter 3, a description of the necessary background theory is given. This theory is included to provide the reader with fundamental knowledge of the information presented in the rest of the thesis.

Chapter 4 gives a broad overview of the general system developed during this study. Further details of this implementation is then described in Chapter 5. Additionally, Chapter 6 presents a specific application, or case, for which this system is used. Here, the general system is customized for this specific case, and different tests and results are described, to show that the system meets its specifications. In Chapter 7 the implementation, testing and results for both the general system and the specific application are being discussed. Then, Chapter 8 presents the conclusions of this entire thesis, as well as some suggested ways of further work.

Finally, the appendices present the content of the attached CD (Appendix A), a list of abbreviations (Appendix B), information and tips about running the program (Appendix C) and some additional details about the sewing cell (Appendix D).

Chapter 2

Literature Review

3D visualization can be very useful in the area of robotics. A specific example of this is for teleoperation. By having a 3D visualization tool, a model of the robot (and/or possibly other devices) can be shown to the human operator. Visualization tools also make it possible to augment the real world, by adding different virtual elements. It is often useful to add some extra information to the screen, based on measurements from different kinds of sensors. In addition to this, the 3D image itself is also very descriptive, compared to simply viewing a 2D image or some textual view of the sensor data. Most 3D visualization tools come as a part of a larger robotic platform, which typically consists of a software framework, utilizing the functionality made available by the middleware. This is very useful for a visualization tool, in order to communicate with other parts of the system, such as receiving updated sensor data. Also, this way it might be easier to reuse code for different robots, even though the hardware on these robots are completely different.

For the past decade, a lot of different robotic software frameworks have been developed. Some of these are very different, and they all have their advantages and disadvantages. Basically, it is impossible to find one framework that is simply the best one for all different robots and scenarios. Examples of this can be that some robots need to have very little functionality to be fast enough, while others need more functionality. Another example is that some of these frameworks might

be aimed at mobile robots, while others are made for industrial robots. In theory, you can use the framework for both applications, but doing this might require more work.

This chapter starts by introducing some of the available robotic software frameworks (Section 2.1). Then, some of the currently available 3D visualization tools are presented in Section 2.2.

2.1 Robotic Software Frameworks

Software frameworks and middleware are often referred to as being almost the same thing. As explained by Schmidt [9], *middleware* is actually low-level software, that can significantly increase reuse, by providing standard solutions to common programming tasks, such as concurrency control and message passing. Moreover, a *framework* is a concrete realization that uses the middleware functionality, in order to provide solutions that can easily be adapted to a specific application. In this section, both frameworks and middleware will be presented, as both are needed to have a good starting point for developing robotic software. There are a lot of different robotic software frameworks available, and this section aims at presenting some of the most commonly used ones. As the goal is to have a robotic platform for different kinds of robots (both mobile and industrial), all kinds of frameworks are considered.

In [4], Kramer et al. present a survey of different available development environments for autonomous mobile robots, including *CARMEN* [10], *Player* [11] and *Pyro* [12]. In addition to these, there are several different robotic frameworks developed by different universities around the world, such as *Robbie* [13]. Other frameworks include *Microsoft Robotics Studio* [14] and the recently popular *Robot Operating System* [2, 3].

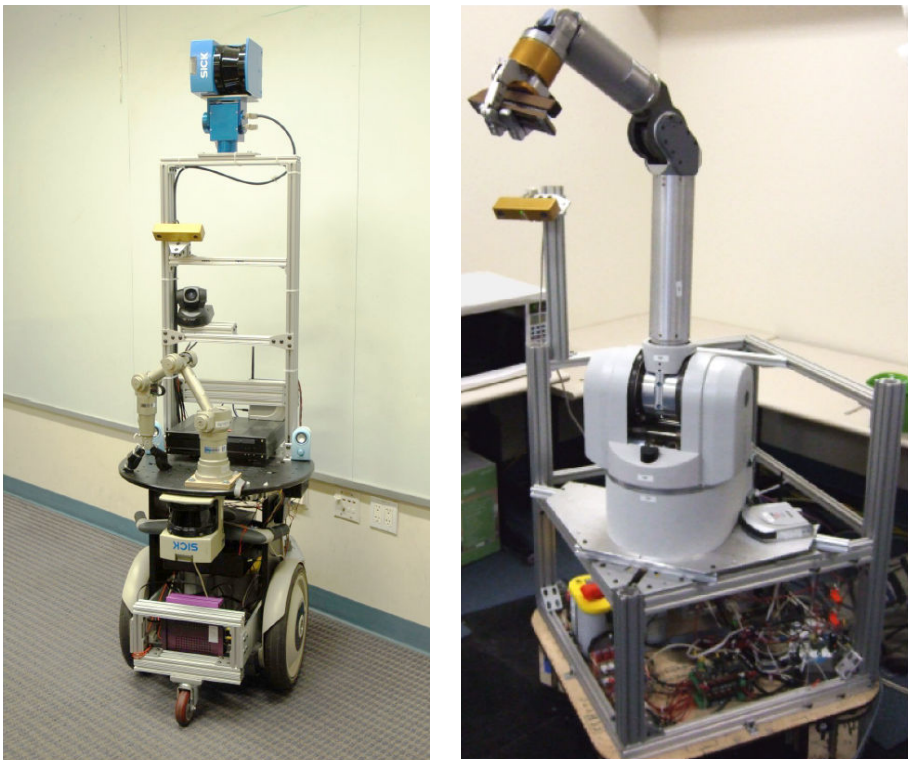
2.1.1 Robot Operating System (ROS)

Robot Operating System (ROS) was originally developed by the Stanford Artificial Intelligence Laboratory [15, 16], and has been under further development by the Californian company Willow Garage [17]. Originally, ROS was created to satisfy needs for the STAIR project at Stanford University, and the Personal Robots Program [18] at Willow Garage. The STanford Artificial Intelligence Robot (STAIR) project is a long-term group effort at the Stanford University, aimed at developing home and office assistance robots [19]. In [19], Quigley et al. describe the first two robots developed as a part of this project: STAIR 1 and STAIR 2. These two robots are fairly different, but they both include a manipulator arm mounted on a mobile base (see Figure 2.1). As shown in this figure, STAIR 1 consists of a Segway base, stabilized with an additional wheel (for safety reasons), and several sensors, including a stereo camera, a pan-tilt-zoom (PTZ) camera and two SICK laser scanners[1], to obtain 3D data about the environment. STAIR 2, on the other hand, consists of a Barrett WAM arm[2], mounted on top of a wheeled base. The base consists of four steerable turrets with two independently-driven wheels, which makes it possible to rotate the robot, as well as moving it in any direction.

The Personal Robots Program is a project, in which the focus is to create robust and safe robots that acts as humans [18]. In this project, Wyrobek et al. have focused on developing an open framework for this purpose. This started off by creating the Personal Robot PR-1 (see Figure 2.2). The PR-1 hardware consists of a 2 Degrees of Freedom (DoF) base, and two 7 DoF arms. The base has two pneumatic-tire wheels attached, with enough force for an 8° climb, and it is able to bump over 1-2 cm obstacles. In addition, the arms have a 5 kg payload each, and can be positioned in similar ways to human arms, to make it as human-like as possible.

Experiences from the software for these projects are the basis for ROS. As mentioned in [3], the philosophical goals of ROS are: peer-to-peer topology, tools-based, multi-lingual, thin, free and open-source. By using a peer-to-peer topology,

[1]SICK laser scanner: http://www.sick.com/
[2]Barrett WAM arm: http://www.barrett.com/



(a) STAIR 1. (b) STAIR 2.

Figure 2.1: Image from [19], showing both STAIR 1 (a) and STAIR 2 (b).

Figure 2.2: Personal Robot PR-1 (image from [18]).

the different nodes (processes) using ROS are able to communicate directly with each other. An alternative way is to use a central server that controls all communication. However, this could lead to more delay in the system, and thus, ROS simply uses a *master node* to set up the different communication links. The *peer-to-peer topology* was also used in the STAIR software. Here, a lot of the communication was through wireless links, which could cause relatively large delays if all traffic was sent through the master. Furthermore, *tools-based* means that all kinds of small tools should be run outside the master, to assure stability and low complexity. Similarly, the idea behind being *thin* is to offer an easy way to extract code, by keeping drivers and algorithms in standalone libraries with no ROS dependencies. As opposed to the software for STAIR and PR, which only supports C++, ROS is *multi-lingual* – supporting C++, Python and LISP. ROS is also *free* and *open-source*, to utilize help from the community to make the software framework as good and complete as possible.

After the release of ROS, a lot of other robotic projects have also ended up using ROS. This includes both newly started projects, and existing projects migrating

to ROS. An example of this is for the robotics team at INRIA Rhône-Alpes. They originally used a self-made middleware called *Hugr* [20]. However, due to difficulties in maintaining and expanding this middleware, they decided to use ROS instead. The reasons why they migrated to ROS, include first of all the lack of manpower to maintain Hugr. In addition, to use other, third party modules, the code would have to be re-written to support the rest of the software. Thus, a much better alternative is to use a software framework also used by a lot of other people. The same argumentation is also used by other robotics groups, and it is one of the reasons why ROS has become quite large in the past 1-2 years. By using a common platform, certain hardware drivers might already be (partly) available, which means all different people using this hardware do not have to re-write the code by themselves.

2.1.2 CARMEN

The Carnegie Mellon Robot Navigation (CARMEN) Toolkit is an open-source robotic framework, written in the C programming language [4]. The main goals of this framework are "to lower the barrier to implementing new algorithms on real and simulated robots, and to facilitate sharing of research and algorithms between different institutions" [21]. To achieve this, CARMEN is organized in a three-layer architecture. The first layer deals with hardware interaction and control, through sensor and motion interfaces. The second layer ("navigation layer") provides different navigation tools, such as motion planning, object tracking and localization. Finally, the third layer is for user applications, utilizing the tools from the second layer.

To assure modularity, CARMEN uses the Inter-Process Communication (IPC) communication protocol. IPC uses TCP/IP, and this makes it possible for different modules to communicate both locally and across a network. Instead of simply running everything on the same CPU, IPC opens up for running modules on different CPUs/computers, which results in a more reliable and extensible system.

Through later updates, CARMEN now also provides support for the Java pro-

gramming language [10]. It also comes with an additional set of tools, such as a simulator, logging, graphical displays and editors.

2.1.3 Player

Player is an open-source project, aimed at multiple-robot, distributed-robot and sensor network systems [22]. According to the authors "the Player robot server is probably the most widely used robot control interface in the world" [11]. *Player* is the robot device server, that assures communication between devices and *clients* in the system. A client connects to Player through a (separate) TCP socket, and then communicates with the devices by sending messages to the server. Originally, Linux was the only supported platform for Player. However, the transition to CMake opened up for multi-platform support, which means it can now also run on Mac OS X and Windows.

The people behind Player has also developed two robot simulators, called *Stage* [11] and *Gazebo* [23]. They are both capable of simulating a population of robots, sensors and other objects. To make the simulated devices as realistic as possible, they are accessed through Player, in the same way as real hardware. The first, Stage, uses a bitmapped 2D environment, while Gazebo uses a 3D world. Lately, Gazebo has also been integrated in other robotic frameworks, such as ROS.

2.1.4 Pyro

Pyro, or "Python Robotics", is another open-source robotics framework [12]. The main goal of this framework is to have an easy-to-understand programming environment, which can be used for educational purposes. Through the use of the Python programming language, as well as low-level abstraction, it is meant to serve as a good starting point for introducing students to the area of robotics. The abstraction includes range sensors, robot units, sensor groups, motion control and devices [4, 24]. For instance, instead of dealing directly with the hardware for sensors such as IR, sonar or laser, they are considered simply as range sensors to the programmer. This provides easier integration in a control program or

something similar. In addition to this, Pyro has also the possibility to integrate C/C++ code. This includes a "wrapping" of Player/Stage (see Section 2.1.3), which makes it possible to utilize the different tools provided by Player/Stage. In addition, Python is a language which is fairly easy to understand, as it looks a lot like pseudo code. Finally, it is also possible to run Python on a lot of different platforms. Thus, because of all the arguments presented above, Python was chosen as programming language, even though it is an interpreted language, which makes it slower compared to other languages.

2.1.5 Microsoft Robotics Studio

Microsoft Robotics Studio (MSRS) is a robotic framework developed by Microsoft, and was first released in 2006 [14]. The architectural goals of MSRS are modularity and a distributed messaging system. This way, decoupled devices (here called services) can easily reuse the same code, and communicate with each other through a network. To make communication between services fast, it is divided into three levels. The first level is for services running on the same processing node. In this case, there is no need to send the data through a TCP socket or some other communication link. Instead, the messages are transferred locally, removing some of the overhead caused by further packet encapsulation. The two other levels uses TCP and TCP/IP with SOAP encoding. This adds a little overhead, but makes it possible to send data through some sort of a network. Furthermore, similarly to Pyro, MSRS uses abstraction to hide low-level details, such as drivers and communication with motors, as well as cameras and other kinds of sensors. This way, programs can be developed, without dealing with the hardware explicitly.

Like other Microsoft developer tools, MSRS is written in .NET. Thus, the main language to use for developing MSRS services is C#. Other alternatives are Visual Basic, Managed C++, IronPython and a visual programming language (VPL), which is included as a part of MSRS. Because .NET is used, the only supported platform is Windows. This might be a problem, as not all devices (robots) support Windows and MSRS. Additionally, due to the fact that the memory management is handled by .NET, there is a potential problem with interruptions, which could

cause unacceptable behavior for a real-time system. MSRS also requires a full .NET library on all services. However, all robots might not have the necessary resources for this. On the positive side, MSRS offers a nice set of tools, such as a 3D simulator using DirectX. Also, it offers an easy way to integrate with other Microsoft products like Kinect and the speech recognition software.

2.1.6 Robbie

Robbie is described by the authors as a message-based robot architecture for autonomous mobile systems [13], and it is developed at the University of Koblenz-Landau. The main design goals of Robbie include a modular and easily maintainable structure, to easily adapt to new hardware. In addition, the architecture supports concurrency, by making sure that each module runs in its own thread. Furthermore, the modules can be configured dynamically, and the complete system is designed to be easy to use, for instance by the use of different abstraction layers. This makes Robbie an easily usable framework for students. Finally, Robbie has a strictly message-based architecture, which means that all communication between modules happens by sending messages. These messages are sent from one module, through the system core, to another module.

Robbie is used in a lot of different projects at the University of Koblenz-Landau, as can be seen in Figure 2.3. Examples are contributions to various robotic competitions, such as *RoboCup@Home* [25], the *RoboCupRescue league* [26] and the *SICK Robot Day* [27], in which they won several awards.

Further extensions made to Robbie, also make it possible to communicate with ROS nodes. This was extension was included in order to still be able to use the old Robbie functionalities, and in addition, utilize some of the ROS functionalities, such as the *RViz* visualization tool (see Section 2.2.1).

2.1.7 Summary

An overview of the different frameworks can be summarized by Table 2.1.

	ROS	**CARMEN**	**Player**
Platforms	Stable: Ubuntu; Experimental: Mac OS X, Windows, other Linux distributions	Linux	Linux, Mac OS X, Windows
Topology	peer-to-peer	central server	central server
Languages	C++, Python, Octave, LISP	C, Java	C, C++, Python, Ruby
Open-source	Yes	Yes	Yes
3D Visualization	Yes (RViz)	No	Yes (Gazebo)
	Pyro	**Microsoft Robotics Studio**	**Robbie**
Platforms	Any (with support for python)	Windows	Linux
Topology	central server (requires Player/Stage)	peer-to-peer	central server
Languages	Python	.NET	C++
Open-source	Yes	No	No
3D Visualization	Yes (requires Gazebo)	Yes	Yes

Table 2.1: A summary of the different robotic frameworks that are presented in this section.

Figure 2.3: Different robots developed at the University of Koblenz-Landau, using Robbie (image from [13]).

Based on the information presented in this section, it can be seen that ROS is a more general framework than the others. It has cross-platform support, peer-to-peer communication, is multi-lingual and open-source, and in addition, contains a lot of useful tools (such as RViz). Based on the performance results presented in [20], it can be seen that ROS typically is a bit slower than some of the others frameworks. This overhead is caused by the additional functionality, which is useful and necessary in many cases.

2.2 Real-time 3D Visualization

In the area of robotics, some sort of 3D visualization tool is useful for a lot of different reasons. First of all, based on sensor data from the robots/environment, the real world can be recreated on a computer – in real-time. This makes it possible to, for instance, view the scene from different angles not accessible in the real world. In addition, this also makes it possible to combine the real data with different virtual objects, such as a line indicating the desired robot trajectory, to see if the robot is able to follow it. Furthermore, this also opens up for viewing the scene at a later time, not only as it actually happens. For instance, 3D visualization tools can be used for debugging, by playing back previously recorded sensor data.

Being able to recreate the real world on a computer can also be important in teleoperation. Using 3D visualization tools make it possible for the operator to see things not directly visible from the available sensors. An example of this is presented in [28]. Here, Huber et al. describe a way to create a photo-realistic 3D model of a real world environment, which is to be used for teleoperation of vehicles. One of these vehicles can be seen in Figure 2.4(a). To generate the 3D model, data from a laser scanner and a video camera attached to the vehicle are combined. Instead of simply controlling the vehicle by looking at the 2D camera view directly, the operator now has a view of the complete 3D scene, and the possibility to view the vehicle/environment from different angles, which makes teleoperation a lot easier. An alternative approach would be to combine data from several cameras to create a 3D view of the environment. However, this would require more data to be transferred and processed, which is not optimal for a real-time application.

(a) (b)

Figure 2.4: Images showing the real tele-operated vehicle (a) and the 3D model (b) (copied from [28]).

2.2.1 RViz

RViz is a 3D visualization environment for ROS [5]. It is tightly integrated with the rest of the ROS tools, and it can be used to visualize a lot of different display types. Examples here are point clouds, robot models, maps, transforms [29], camera/image displays and markers. A lot of different display types can then be combined in

one view, to create a complete picture of a specific scenario. In addition to simply viewing the different display types, RViz is also able to give feedback to some of these, such as interactive markers. From within RViz, the interactive markers can be moved around in space, clicked, or edited through a menu [30]. The events will then be sent through the interactive marker server, and back to the application in which the marker was created. A more detailed description of RViz can be found in Chapter 3.

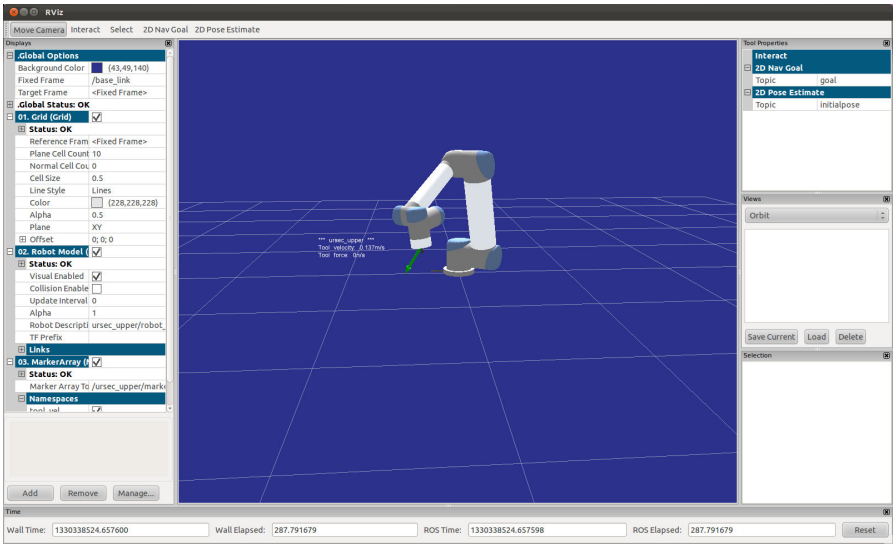


Figure 2.5: Example of an RViz configuration, with one robot (Universal Robots UR-6-85-5-A), and two markers (text and an arrow to indicate the tool velocity).

RViz has been used for a lot of different applications. The typical scenario is having a robotic platform running ROS, and using RViz to provide a 3D visualization of the environment, which for instance can also include the motion trajectory. An example of this is presented in [6]. Here, Ferland et al. have created an omnidirectional non-holonomic robotic platform, called AZIMUT-3 (see Figure 2.6). This robot is controlled by a simple joystick, through the use of ROS. RViz is here used for a real-time display of the commanded instantaneous center of rotation (ICR).

Furthermore, in [7], Jayasekara et al. present a system for localizing and tracking multiple mobile robots. The robots are localized using a laser range finder and a camera based sensor unit. Then, the robots are tracked using the Rao-

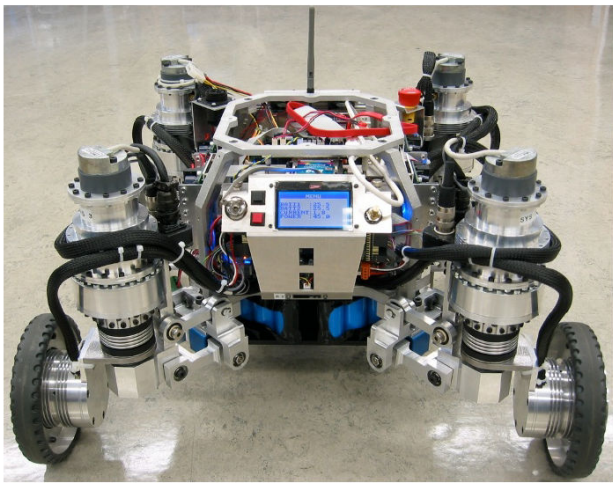
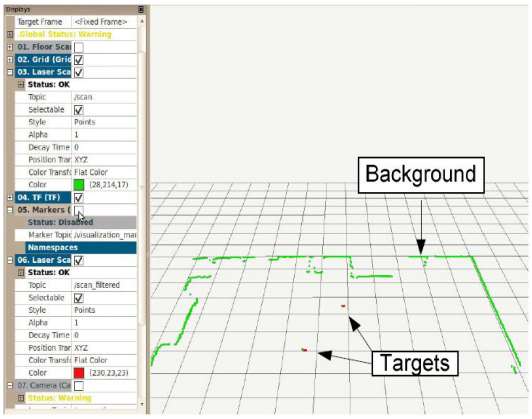
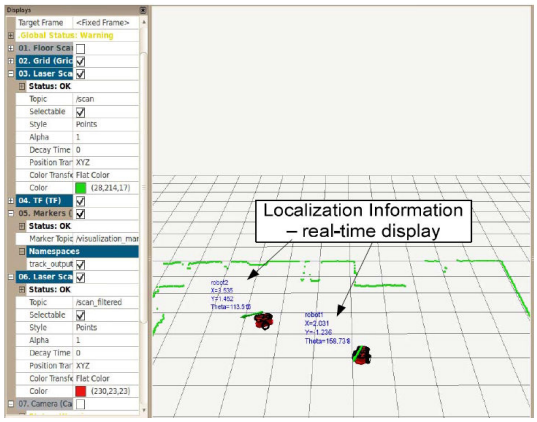


Figure 2.6: AZIMUT-3 – the omnidirectional non-holonomic robotic platform. Image from [6].

Blackwellized particle filter technique. Here, RViz is used to display localization information for the mobile robots – in real-time. The resulting RViz display can be seen in Figure 2.7.



(a) Raw data from the laser range finder.



(b) Localization information.

Figure 2.7: This figure shows the raw data from the sensor, as well as the added localization information for the targets. Image from [7].

Chapter 3

Background Theory

This chapter presents the necessary background theory for the rest of the thesis. Firstly, Section 3.1 gives a brief description of Robot Operating System (ROS), and some of the included tools. Secondly, Section 3.2 presents some different ways to represent transformations of coordinate systems. Finally, Section 3.3 describes both hardware and software of the Kinect sensor.

3.1 Robot Operating System (ROS)

ROS is a software framework for robot applications created and maintained by the company called Willow Garage [17]. The general information about ROS presented in this section is copied from the project report [31], with some modifications/additions.

As stated on the official ROS website [2]:

> *ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.*

In other words, ROS has two main functions; one to provide operating system functionalities, and the other to provide user contributed packages for different kind of robot applications.

As stated above, ROS is an open source, meta-operating system. This means, it is not a stand-alone operating system, but it has to run on top of another operating system. Currently, there exists a stable version for Ubuntu, and experimental versions for Max OS X, Arch, Fedora, Gentoo, OpenSUSE, Slackware, Debian and Windows [2]. ROS has really gained popularity the last few years. As an example, the total number of available open-source repositories doubled from November 2010 to May 2011. Currently, ROS supports at least the three following languages: C++, Python and LISP [3]. In addition, the developers are working on support for other languages, such as Java and Lua.

To understand how ROS works, there are a few concepts and expressions that has to be explained. First of all, a *node* is the part of a program doing the actual computation (usually referred to as the process). Typically, one would run several nodes together in a large system, each doing some specific work. When running ROS, there must always be a *master* node running. The master is providing name registration and lookup for all nodes, which makes it possible for the other nodes to communicate with each other. To communicate, a connection between the different nodes are set up by the master node, and the nodes then communicate directly with each other using *messages*. Messages consist of one or more data types with information (such as strings, integers, etc.). Furthermore, there are mainly two ways to send these messages between two (or more) nodes. The first is by using *topics*, which can be thought of as some sort of a communication bus. This way, nodes can subscribe or publish messages to a topic, where subscribing means to listen for messages, and publishing means to send out messages. By subscribing to a topic, the node will receive a callback when new messages are being published to the topic by other nodes. The other way for nodes to share messages, is through *services*. Here, you have a service node, and one or more client nodes. The general way of using services, is for the client nodes to make a request, and wait for an answer from the service node. The service node, on the other hand, first advertises its service and then waits for other nodes to request it.

Nodes can also store data on something called the *parameter server*. The parameter server is actually a part of the master node, and it provides a way for nodes to store data by key. These parameters can be stored and retrieved by any node at runtime. Typically, parameters are used to store configurations, so that the different nodes easily can view these while running.

An additional concept for understanding the structure of ROS is a *package*. A package is the main part for organizing the software. It may consist of nodes (processes), message or service definitions, libraries and so on. All packages must also contain a manifest (an *.xml* file) with meta-data about the package. The manifest contains license information, package dependencies, the name of the author(s), and a package description. In addition to this, ROS also uses *stacks*. A stack is a collection of different packages. Typically, the packages in a stack work in conjunction with each other. An example is the `joystick_drivers` stack. This stack consists of several packages, where the different packages contain drivers for different joysticks.

Through all the different available stacks, ROS provides a lot of useful tools for robotic applications. These tools include drivers for different robots and sensors, as well as tools for 3D visualization (RViz), real-time plotting (Rxplot), communication graphs (Rxgraph), etc. Another example ROS tool is *Rosbag* [32]. Rosbag makes it possible to record and play back data on topics. Data is recorded by subscribing to the topics of interest, and saving the information as *.bag* files (*bags*). This data can be played back in correct time intervals at a later time.

Furthermore, ROS also includes wrappers for both Point Cloud Library (PCL) [33] and OpenCV (Open source Computer Vision) [34]. PCL and OpenCV are open-source libraries for 3D point cloud processing and real-time computer vision, respectively. Both of these libraries are developed by Willow Garage, and they are very useful in many robot applications.

3.1.1 RViz

RViz is a 3D visualization tool for ROS [5]. Using this program makes it possible to visualize a lot of different display types. These can be viewed alone, or together with other displays types. All available display types are listed and described in Table 3.1.

An example RViz configuration can, for instance, consist of a *point cloud* combined with different *markers*. The result of this is an augmented view of the real world. The markers can be basic shapes, such as arrows, cubes, spheres, cylinders and so on, or more advanced mesh types defined as either *.stl*, Ogre *.mesh* or COLLADA (*.dae*). As presented in Section 2.2.1, RViz also supports interactive markers. These can be moved around in space, clicked or edited through a menu in RViz. The events will then be sent back to the application that created the specific marker, through an interactive marker server. The visual primitives used for interactive markers are the same as for the regular markers.

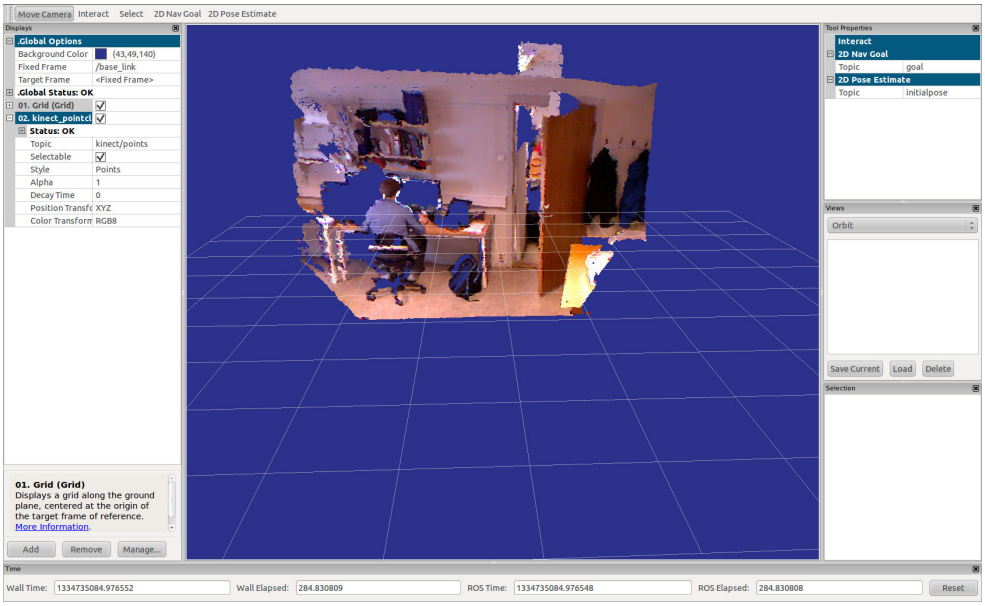


Figure 3.1: Example screenshot of RViz, showing a point cloud generated by data from a Kinect sensor.

Another example usage of RViz can be to combine a *robot model*, a *map* and a *laser scan* display type. Then, as the robot moves around, the global map of feasible

Name	Description
Axes	Displays a set of Axes
Camera	Creates a new rendering window from the perspective of a camera
Grid	Displays a 2D or 3D grid along a plane
Grid Cells	Draws cells from a grid, usually obstacles from a costmap from the navigation stack.
Image	Creates a new rendering window with an Image. Unlike the Camera display, this display does not use a CameraInfo.
Interactive Marker	Displays 3D objects from one or multiple Interactive Marker servers and allows mouse interaction with them.
Laser Scan	Shows data from a laser scan, with different options for rendering modes, accumulation, etc.
Map	Displays a map on the ground plane.
Markers	Allows programmers to display arbitrary primitive shapes through a topic.
Path	Shows a path from the navigation stack.
Pose	Draws a pose as either an arrow or axes.
Pose Array	Draws a "cloud" of arrows, one for each pose in a pose array.
Point Cloud(2)	Shows data from a point cloud, with different options for rendering modes, accumulation, etc.
Polygon	Draws the outline of a polygon as lines.
Odometry	Accumulates odometry poses over time.
Range	Displays cones representing range measurements from sonar or IR range sensors.
Robot Model	Shows a visual representation of a robot in the correct pose (as defined by the current TF transforms).
TF	Displays the `tf` transform hierarchy.

Table 3.1: List of the built-in display types in RViz (copied from the RViz User Guide [5])

areas for the robot can be seen. Additionally, data from a laser scanner can be added, to show more clearly an estimate of the (virtual) reality.

When combining views of different display types in RViz, it is important to consider how the different coordinate systems relate to each other. RViz uses the *tf* transform system, which is described in more detail in Section 3.1.2. By using tf, RViz is able to transform the coordinate frames of the different displays into a global reference frame. Choosing which frame to use as the global reference frame, can easily be done as a part of the configuration.

3.1.2 Tf

Tf is a transform system used to keep track of the relation between different coordinate frames in ROS [29]. It is implemented as a ROS package called `tf`, which contains an API for programming in both C++ and Python. The relationship between the coordinate frames are maintained in a tree structure. As an example, a robot has typically a lot of coordinate frames, such as a base frame, arm frames, head frame, and so on, all of which are relative to some fixed global frame. Figure 3.2 shows a robotic system with several coordinate frames. An example overview of the relation between some of the different frames are shown in Figure 3.3.

3.1.3 URDF

Unified Robot Description Format (URDF) is an XML format for representing robot models [35]. It is developed for usage under ROS, and is a part of the `robot_model` ROS stack. This stack also provides different tools for URDF files, such as a C++ parser and a validator.

The URDF XML syntax should be relatively intuitive for most people. Listing 3.1 shows a sample implementation of the robot in Figure 3.4. As shown here, a URDF file uses a tree structure, with `<robot>` as the root element. This element then contains several `<link>`s, connected together with `<joint>`s. Different features of

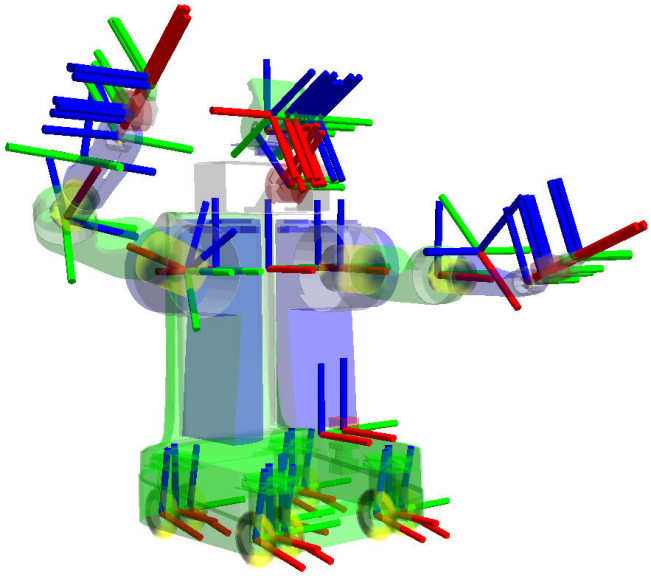


Figure 3.2: An example robot, with several different coordinate frames (copied from [29]). This is actually the PR2 robot, which is a descendant of PR1 (presented in Section 2.1.1).

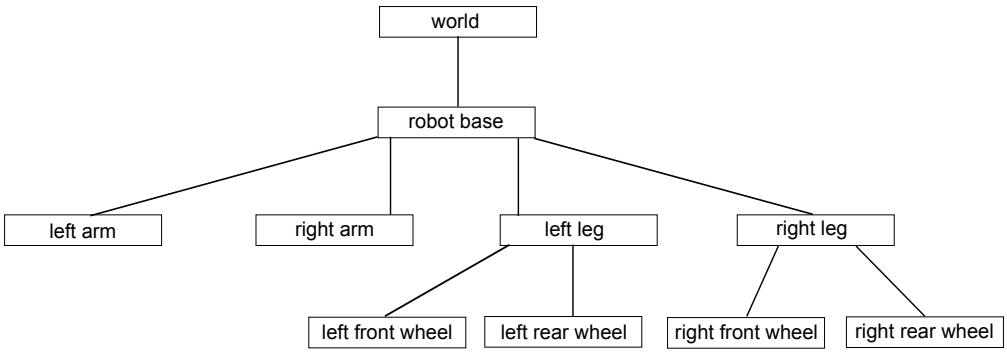


Figure 3.3: An example of the relation between different coordinate frames.

the links and joints can be further described by other elements, such as `<visual>` and `<collision>` for links and `<origin>` and `<axis>` for joints. A more detailed description of URDF can be found on the ROS wiki page [35].

```
<robot name="sample_robot">
    <link name="link1" />
    <link name="link2" />
    <link name="link3" />
    <link name="link4" />

    <joint name="joint1" type="continuous">
        <parent link="link1"/>
        <child link="link2"/>
    </joint>

    <joint name="joint2" type="continuous">
        <parent link="link1"/>
        <child link="link3"/>
    </joint>

    <joint name="joint3" type="continuous">
        <parent link="link3"/>
        <child link="link4"/>
    </joint>
</robot>
```

Listing 3.1: Example of a very simple robot (Figure 3.4) described in the URDF XML language

Xacro

Xacro is a macro language used to simplify XML files. It is a powerful ROS package, that is used in many applications, for instance for URDF files. By using Xacro, the robot description files can be much easier maintained, by increasing the readability and easier reuse of code snippets. As described on the ROS wiki [36], Xacro has several different features. First of all, the main feature is the support for macros. By using the macro tag, large pieces of code can easily be reused. An

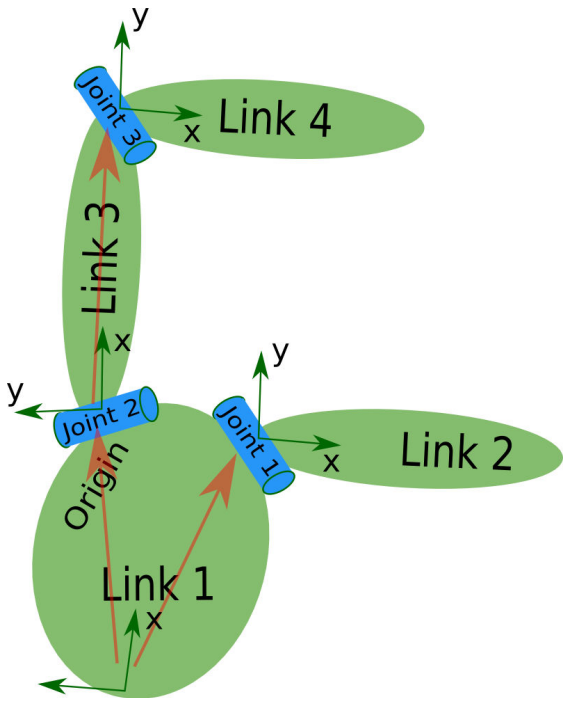


Figure 3.4: Example robot with 4 links and 3 joints (copied from [35]).

example of this can be seen in Listing 3.2. In addition to this, Xacro also supports basic math expressions and other properties (Listing 3.3).

```
<xacro:macro name="robot_arm" params="prefix rotation">
    <link name="${suffix}_arm" />
    <joint name="${suffix}_arm_joint">
        <origin rpy="0 0 ${rotation}" />
        ...
    </joint>
</xacro:macro>
<xacro:robot_arm prefix="left" rotation="0" />
<xacro:robot_arm prefix="right" rotation="1.570796" />

<!-- This would be equivalent to the following: -->

<link name="left_arm" />
<joint name="left_arm_joint">
    <origin rpy="0 0 0" />
    ...
```

```
</joint>

<link name="right_arm" />
<joint name="right_arm_joint">
    <origin rpy="0 0 1.570796" />
    ...
</joint>
```

Listing 3.2: Using xacro to create a simpler URDF file.

```
<xacro:property name="pi" value="3.14159" />
<xacro:property name="radius" value="2.5" />

<circle circumference="${2 * radius * pi}" />
```

Listing 3.3: Xacro example, using properties and math expressions.

3.2 Transformations of Coordinate Systems

There are several different ways to represent transformations between different coordinate systems, or frames. This section shortly presents different ways to represent rotations, using rotation matrices, Euler angles, axis-angle representation and quaternions. In addition to this, a way to represent relative transformations, using Denavit-Hartenberg (DH) parameters, is also included. DH parameters are common for representing the relationship between adjacent robot links. The information presented in this section is important for modeling robots and understanding how to calibrate different devices in a scene, based on their coordinate frames.

3.2.1 Rotation Matrix

A rotation matrix $\mathbf{R}_b^a$ can have three different interpretations [37, 38]. Firstly, it is an operator used to rotate a vector, to get a new vector in the same coordinate frame. For instance, a vector $\mathbf{p}^a$ in a is rotated to the vector $\mathbf{q}^a$, by $\mathbf{q}^a = \mathbf{R}_b^a \mathbf{p}^a$. Secondly, it represents a coordinate transformation for a point p from frame b

to frame a. That is, $p^a = \mathbf{R}_b^a p^b$. Finally, the rotation matrix also gives relation between a transformed coordinate frame and a fixed coordinate frame.

The most basic rotations in 3D are rotations about a fixed axis. These rotations are called *simple rotations*. Equations (3.1) - (3.3) show what the rotation matrices looks like, for a simple rotation ϕ about x, θ about y, and ψ about z, respectively.

$$\mathbf{R}_x(\phi) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{pmatrix} \tag{3.1}$$

$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix} \tag{3.2}$$

$$\mathbf{R}_z(\psi) = \begin{pmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.3}$$

3.2.2 Euler Angles

There are several different ways to use rotation matrices to represent a specific rotation. One of these is by using *Euler angles*. Euler angles are given as three parameters: ϕ, θ and ψ, which represents the rotation about three axes. The axes can be any permutation of the x-, y- and z-axis. For instance, Figure 3.5 shows a ZYZ-Euler angle transformation. Here, the first rotation is about the z-axis by the angle ϕ, followed by a rotation about the new y-axis by the angle θ. The final rotation is about the current z-axis by the angle ψ. This rotation can be written as:

$$\mathbf{R}_b^a = \mathbf{R}_{Z,Y,Z} = \mathbf{R}_{z,\phi}\mathbf{R}_{y,\theta}\mathbf{R}_{z,\psi} \tag{3.4}$$

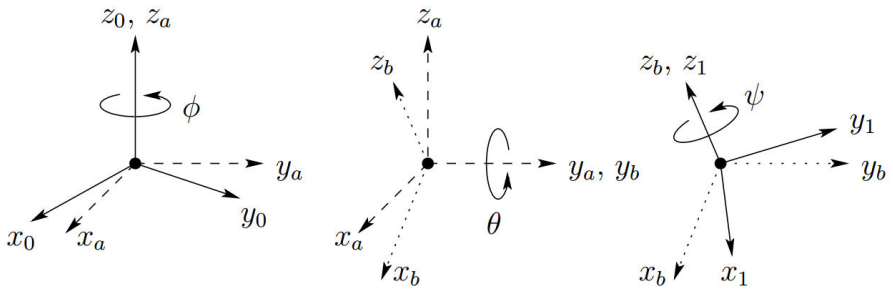


Figure 3.5: A ZYZ-Euler angle transformation. Image from [37].

3.2.3 Axis-Angle Representation

Another way to describe a rotation with only a few parameters, is to use *axis-angle representation*. The axis-angle representation of a rotation is described by a unit vector $\mathbf{k} = (k_x,\ k_y,\ k_z)^T$ indicating the axis of rotation, and an angle θ describing the magnitude of rotation about $\mathbf{k}$. Figure 3.6 illustrates how this works. In this figure, α and β are the angles of rotations about x and y needed to align the z-axis of the initial coordinate frame to $\mathbf{k}$.

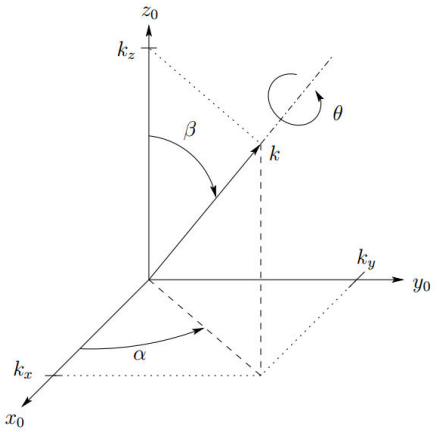


Figure 3.6: Axis-angle representation of a rotation. Image from [37].

3.2.4 Quaternions

A quaternion is yet another way to describe a rotation. A quaternion $\mathbf{q}$ consists of a scalar part α and a vector part $\boldsymbol{\beta}$. Following the notation used in the different

ROS packages, a quaternion is written with the vector part first, then the scalar part. That is, it can be written as:

$$\mathbf{q} = \alpha + i\beta_1 + j\beta_2 + k\beta_3 \tag{3.5}$$

$$= \begin{pmatrix} \boldsymbol{\beta} \\ \alpha \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \alpha \end{pmatrix} \tag{3.6}$$

An alternative way to describe a quaternion is using $\boldsymbol{\beta} = (x,\ y,\ z)^T$ and $\alpha = w$:

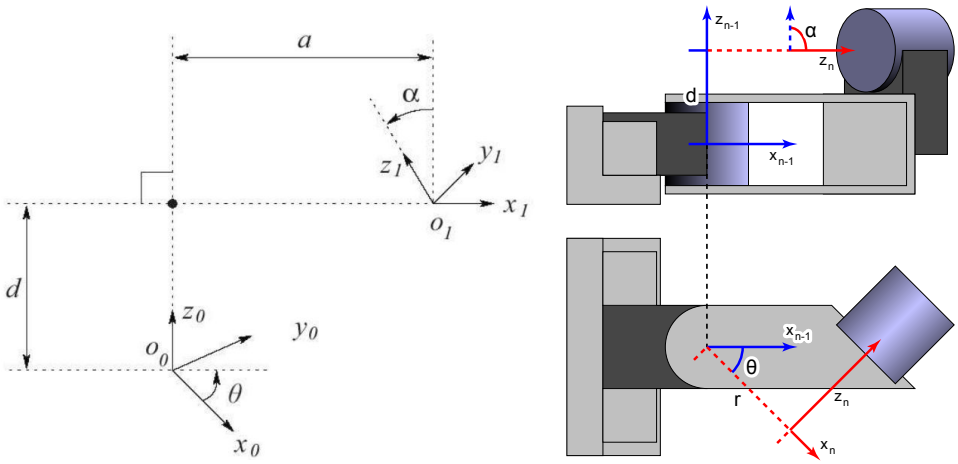
$$\mathbf{q} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{3.7}$$

More details about quaternions can be found in [38].

3.2.5 Denavit-Hartenberg Parameters

The Denavit-Hartenberg (DH) convention is a way to represent a homogeneous transformation between two coordinate frames. This convention uses four parameters, called DH parameters, which for a link i are d_i, θ_i, a_i, and α_i. Here, d_i is the offset along the previous z-axis to the common normal [39]. θ_i is the angle about the previous z-axis, from the old x-axis to the new x-axis. Furthermore, a_i is the length of the common normal, and α_i is the angle about the common normal, from old z-axis to the new z-axis. These parameters are illustrated in Figure 3.7. Note that in Figure 3.7(b), the parameter a is actually called r.

There are two conditions that have to be satisfied, in order to represent a homogeneous transformation using only these four parameters. First, the new x-axis must be perpendicular to the previous z-axis. In addition to this, the new x-axis must also intersect the previous z-axis.



(a) Image from [37]. (b) Image from [39].

Figure 3.7: Two figures illustrating what the different DH parameters represent.

3.3 The Kinect Sensor

The Kinect sensor is created by Microsoft, and released in early November 2010 [8, 40]. It was originally created for gaming purposes, on the Microsoft Xbox 360 console. However, due to the fairly advanced technology and cheap price, it has also become very popular for researchers and scientists. Because of this, 8 million units were sold during the first 60 days, which gave the Kinect the Guinness World Record of being the fastest selling consumer electronics device. As of January 2012, a total of 18 million Kinect devices have been sold world-wide.

3.3.1 Technology

As can be seen in Figure 3.8, the Kinect consists of a depth sensor, RGB camera, and a multi-array microphone. The depth sensor is based on technology by the Israeli company PrimeSense, and consists of an infrared laser projector, combined with a monochrome CMOS sensor. This makes it possible for the sensor to capture depth data from the environment without any specific lighting conditions. To capture range data, the projector illuminates the scene with a laser dot pattern. The depth is then calculated by combining the pattern from the infrared projector

and the pattern captured by the CMOS sensor. The resulting depth data can then be used in combination with data from the RGB sensor to create an informative point cloud with colors.

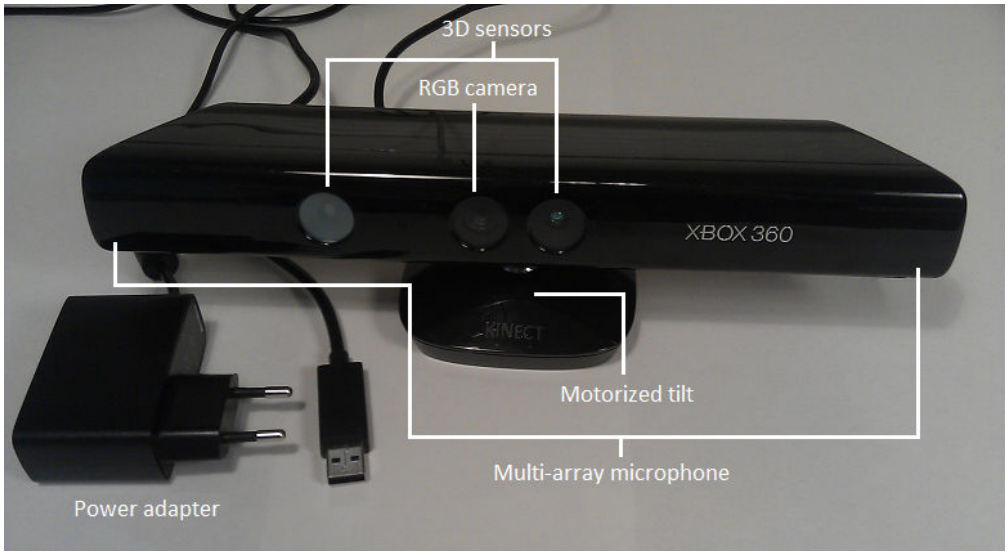


Figure 3.8: Image of a Kinect sensor.

Furthermore, the multi-array microphone can be used together with some voice recognition software, to create applications that interacts with people even more naturally. In addition to this, the Kinect also consists of an accelerometer, as well as a motor, used to control the physical tilt.

Microsoft has not officially published the exact specifications of the Kinect sensor. However, a most of the specifications have been figured out through the use of reversed engineering, and are listed in Table 3.2 [41].

3.3.2 Using the Kinect in ROS

There are two different Kinect drivers available in ROS. The first is based on the *libfreenect* library [42], and is released as a part of the `kinect` ROS stack. This stack is now listed as deprecated, due to the driver being slightly limited and outdated. Libfreenect was the first Kinect driver available to PC and Mac users, and it is developed by the *OpenKinect* community, mainly through the use

Property	**Specification**
Horizontal field of view	57°
Vertical field of view	43°
Physical tilt range	±27°
Depth sensor range	0.8m - 6m
Lateral resolution	1.3mm @ 0.8m
Depth resolution	∼ 1cm
Depth image size	640 x 480 @ 30Hz
Depth image depth	11-bit
RGB image size	640 x 480 @ 30Hz, 1280x960 @ 10Hz
RGB image depth	8-bit, Bayesian color filter
Audio	16-bit audio @ 16kHz
Power	12 W
Connectors	USB 2.0 + Power
Skeletal Tracking System	Tracks up to 6 people simultaneously. However, only 2 active players (with 20 joints per player)
Price	∼ 1000 NOK

Table 3.2: List of Kinect specifications (copied from [41]).

of reversed engineering. In the beginning, this was the only driver available, and it was therefore very popular.

The second Kinect ROS driver is a wrapping of the official OpenNI driver [43], and is a part of the `openni_kinect` stack. This stack also contains wrappings of NITE, which is middleware for naturally interacting with humans. Both OpenNI and NITE are originally created by PrimeSense [44]. In addition to this, there is also a package called `openni_launch` available in ROS. This package launches files to open an OpenNI device, such as the Kinect, as well as nodelets used to convert raw data streams to different objects, like disparity images and point clouds. Using

this launch file also makes it possible to get a registered point cloud (RGBXYZ), where each point has both RGB (color) and XYZ (position) values. For more detailed information about the Kinect sensor, see [31, 40].

Chapter 4

System Overview

The complete system created as part of this thesis can be seen in Figure 4.1. Here, the solid line objects represent the general system components, while the dashed line objects represent application specific devices. In addition, the arrows show the typical communication flow between the different components. In this figure, an application specific device is illustrated as a 'black box', consisting of the hardware and software necessary to communicate with the rest of the system. By sending out information according to what is shown for the communication links in the figure, these devices are seamlessly connected to the rest of the general system. In Figure 4.1, one robot, one depth sensor and one 'other' device is shown. However, the system is designed so that there can be any number of either device. There can, for instance, be two robots and two depth sensors. Each of the robots will then have a *Robot Data Processor* node associated with it, and similarly for the depth sensors. To make it possible to select different devices to use, a configuration file has to be specified as input to the system. This configuration file uses XML syntax, and contains information about all the available devices, such as what its base frame is, and what other nodes it requires to run successfully.

The idea behind all of this is to have a system that is able to visualize a process consisting of robots and other devices, both in online – real-time – and offline. To do this, ROS and RViz was chosen as a starting point. As explained in Chapter 3, Robot Operating System (ROS) is a robotic software framework. ROS contains

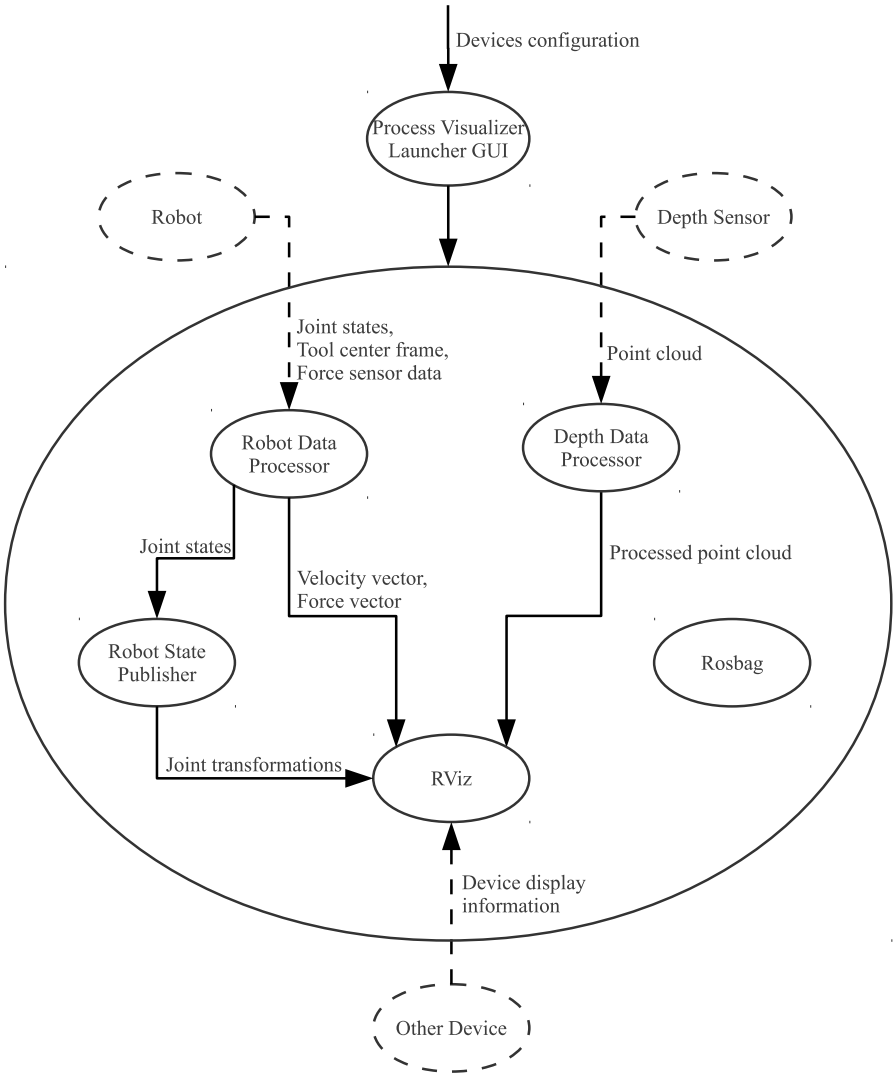


Figure 4.1: An overview of the complete system.

a lot of tools that are useful when developing robot applications. One of these tools is the visualization environment RViz. RViz can be used to visualize a lot of different devices, and was therefore chosen as the basis for the system described in this thesis. In RViz, several different devices can easily be combined in a virtual scene, and be viewed from all possible angles.

In the following sections, the different system components shown in Figure 4.1 are presented. A more detailed description of the system implementation is described in Chapter 5. The information presented in this chapter is for the general part of this system. To actually be able to use this system, a specific case or application with real sensors and devices is needed. Chapter 6 explains how this system is used in a real application – the Ekornes sewing cell at SINTEF Raufoss Manufacturing.

4.1 Process Visualizer Launcher GUI

The main part of this system is the *Process Visualizer Launcher GUI*. This is a ROS node, for which the purpose is to have an organized way to control the rest of the system. This node consists of a GUI, used to choose which devices should be displayed, as well as ways to record and playback data. As shown in Figure 4.1, the input to this node is a configuration file. This configuration file should contain information about all the available devices in the system, and must be written by using a specific XML syntax.

Furthermore, this GUI is used to launch the extra ROS nodes associated with the device. What extra nodes to launch for a devices, is specified either in the XML configuration or by the device type. For instance, by default a robot device has a corresponding *Robot Data Processor* and *Robot State Publisher* node, and a depth sensor has a corresponding *Depth Data Processor* node. To actually be displayed in RViz, each of the devices needs one or more display types. These also depend on the device type. For the display types to show up in RViz, an RViz configuration file must be specified. This configuration file is created dynamically by the GUI node, based on the selected devices.

This program has two different modes: *real-time* and *playback* mode. Through the use of the GUI, it is possible to record and play back data, in accordance with the current program mode. That is, in real-time mode the GUI shows an option to record data, while in playback mode it provides functionality for playing back data, using a few media buttons. To control the recording and playback of data, this node sends commands to the Rosbag node. Additionally, when playing back data, it receives the current playback time. The implementation of this node is explained in detail in Chapter 5.

4.2 Robots

As mentioned earlier, the system supports multiple robots. With respect to Figure 4.1, this means multiple robot objects, each with a corresponding *Robot Data Processor* and *Robot State Publisher* node. What is here illustrated as a dashed line object consists of the hardware and low-level software. More specifically, this must include all the parts necessary to send data to the Robot Data Processor node, and its contents are not considered a part of the system described in this chapter. Typically, this box consists of the actual hardware, drivers, and ROS middleware that is used to publish the sensor data on topics.

Robot Data Processor is a node that is used to receive data from robots, and transfer this into information that can be used by RViz. First of all, the Robot Data Processor receives the robot joint states, tool center frame and force sensor data. The joint states vector is then transferred into a `sensor_msgs/JointState` data type. These messages are then sent to the Robot State Publisher, which calculates the current joint transformations (frames). Secondly, the Robot Data Processor receives the tool center frame and force sensor data. If this data is available, it is used to calculate the tool velocity and force, which is to be illustrated as vectors, or arrow markers, in RViz.

4.3 Depth Sensors

Depth sensors are added to the system similarly to robots. There can be multiple depth sensors, each with a corresponding *Depth Data Processor* node. A reason for having multiple depth sensors could be to generate a more detailed point cloud, by viewing the scene from different angles. Also here, the dashed line object indicating a depth sensor contains all parts necessary to generate the point cloud, which is to be processed by the Depth Data Processor node. This includes the hardware, drivers and image processing software.

Depth Data Processor is a very simple node that only updates the timestamp on the point clouds. The reason for having this node, is because it is necessary to update the timestamp on point cloud data offline – in order to be displayed in RViz. Otherwise, RViz would see the data as old, compared to the other data it receives, and would therefore choose not to display the point cloud.

4.4 Other Devices

What is here referred to as an *'other'* device is basically a device that is neither a robot nor a depth sensor. Since this really can be anything, it does not have a common data processing node, such as for robots and depth sensors. Thus, everything that is to be visualized in RViz, must be published by the device itself. As an example, the 'other' device object shown in Figure 4.1 could include the actual hardware and software drivers of the device, as well as a ROS node used to generate some sort of a display type supported by RViz, such as a marker.

4.5 ROS Tools

The rest of the objects shown in Figure 4.1 are other ROS tools and packages used in this system. First of all, as mentioned in Section 3.1, *Rosbag* is a set of tools for recording and playing back data on topics [32]. Even though it is

also possible to use the Rosbag code API, the Rosbag command-line tool is used for this thesis. The main reason for this is that it is simpler, while still providing enough functionality for this implementation. The Rosbag node is started, paused, resumed and stopped by the GUI node, according to the specified configurations. Specifying which topics should be recorded or played back is also done in the GUI node.

Another ROS node that is used in this system shown in Figure 4.1 is *Robot State Publisher* (`robot_state_publisher`). This node subscribes to a topic called `/joint_states`, which contains information about all joint states for a particular robot. The current joint states are then used to create proper transforms (tfs) for the robot frames, which again are published on the `/tf` topic. To successfully accomplish this calculation, the incoming states must be of a specific format (the `sensor_msgs/JointState` data type). Also, for this node to be able to transform all frames, a URDF XML description of the robot must be set on a parameter called `robot_description`. The Robot State Publisher node is used mainly for simplicity. An alternative solution would be to do the same calculation inside the Robot Data Processor node. However, since this tool is already available in ROS, there is no point in writing additional code to do this.

The final main part of this system is *RViz*. This node is the one doing the actual visualization of the process, and it is launched with configurations specified by the selected devices in the Process Visualizer GUI. The different devices displayed in RViz are updated through transforms and other topics published by the data processor and state publisher nodes. The main reason for choosing this node is because of its tight integration with ROS, which makes it fairly easy to combine data from different sensors into one visualized 3D scene.

Chapter 5

Implementation

This chapter describes the implementation of the *Process Visualizer*, introduced in the previous chapter. The Process Visualizer contains all the general parts of the system described in this thesis, and it is implemented as a ROS package called `process_visualizer`. More specifically, this is where all the general parts of the system (shown with solid lines in Figure 4.1) are implemented. The *Launcher GUI*, *Robot Data Processor* and *Depth Data Processor* nodes shown in this figure are created specifically for this master thesis, while *RViz*, *Rosbag* and *Robot State Publisher* are other ROS tools that are used by the rest of the system.

The Process Visualizer is organized as shown in Figure 5.1. This figure shows a diagram of all the classes, implemented in Python, as a part of the Process Visualizer. First of all, the Launcher GUI is implemented as a main class called `LauncherApp`. This class uses four other classes: `BagRecorder`, `BagPlayer`, `RVizConfig` and `XmlParser`. The first three of these are considered a part of the Launcher GUI, by providing functionality for recording and playing back data, as well as dynamically creating the configuration file to use in RViz. In addition, the `XmlParser` class is used to parse the XML configuration files. Furthermore, the Process Visualizer also contains of two more classes: `RobotProcessor` and `DepthProcessor`. These two classes are used to process sensor data from robots and depth sensors.

This chapter starts off by explaining the implementation details of the Process

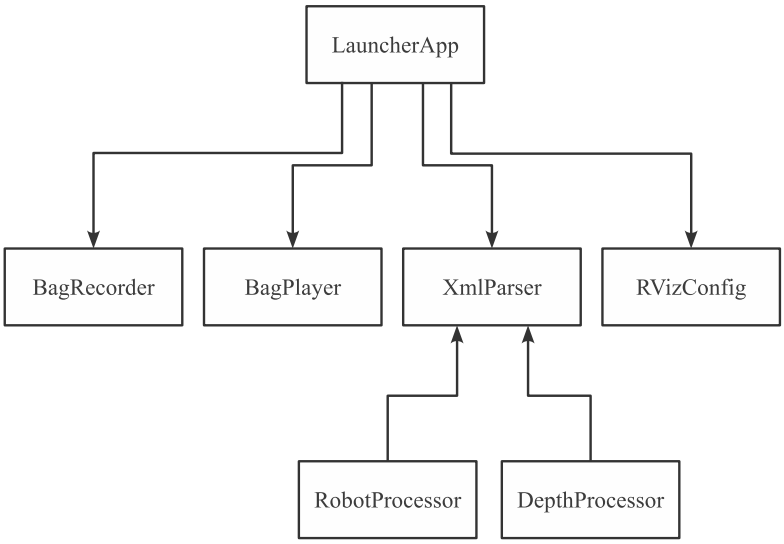


Figure 5.1: Class diagram of the `process_visualizer` package.

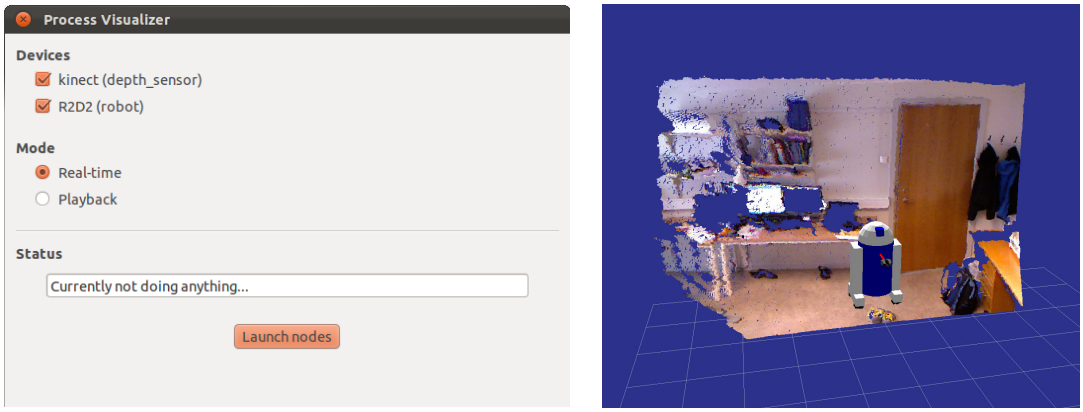
Visualizer Launcher GUI in Section 5.1. Then, both the actual implementation and syntax of the XML configuration is presented (Section 5.2). Finally, in Section 5.3 and 5.4, the Robot and Depth Sensor Data Processor nodes are explained.

5.1 Launcher GUI

Launcher GUI is the main part of the Process Visualizer. This is where all the other nodes are started, based on the configuration specified in the XML file (Section 5.2) and the GUI itself.

At startup, the window in Figure 5.2(a) is shown. This window shows a list of devices, the possible program modes, the current program status and a launch button. The list of *devices* is based on information from the XML configuration file, and is used to indicate which devices should be launched. It is possible to select the desired devices in the GUI, and this information is used to launch the correct nodes, as defined in the configuration file. Furthermore, the *mode* is simply used to specify whether the program should run in real-time or not. Running in real-time mode makes it possible to record data, while running in playback mode makes it possible to play back data from a prerecorded *.bag* file. Finally, the *status*

bar shows information about the current program state, and the *launch button* is used to launch the nodes associated with the selected devices.



(a) Launcher GUI. (b) RViz view of the example.

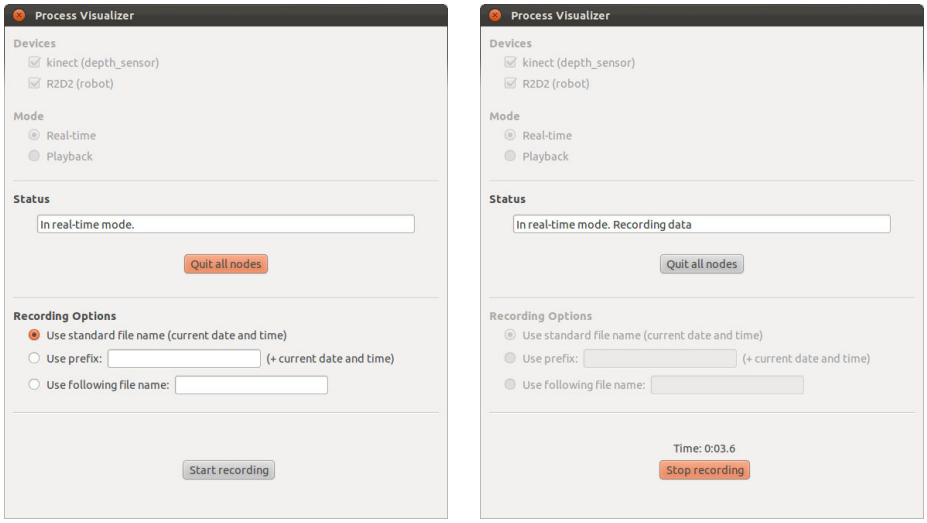
Figure 5.2: Launcher GUI and the corresponding RViz view.

When the launch button is clicked, the window changes, depending on the selected mode. In *real-time* mode, the current view is as shown in Figure 5.3(a), and in *playback* mode, the view is as shown in Figure 5.4(a). These parts of the GUI are implemented in the `BagRecorder` (Section 5.1.1) and `BagPlayer` (Section 5.1.2) classes, respectively, and include functionality for recording and playing back data, using Rosbag. Additionally, clicking the launch button starts several different nodes. For each of the selected devices, all ROS nodes and launch files specified in the XML configuration are launched. The XML configuration for the example shown in Figure 5.2 is presented in Section 5.2.2. For each of the extra nodes and launch files, it is possible to specify whether they should run in real-time mode, playback mode, or both. In addition to this, a ROS node called `static_transform_publisher` is also launched. This node simply calculates and sends out transformations between the global frame and the device's base frame, at a given frequency (default 10 Hz). Furthermore, *Robot Data Processor* (Section 5.3) is started for all robots, and *Depth Data Processor* (Section 5.4) is started for all depth sensors. In addition to this, an RViz configuration file is generated dynamically by an instance of the `RVizConfig` class, based on the selected devices. Then, finally, the actual 3D visualization node, RViz, is launched.

As mentioned in the introduction to this chapter, all code is written in the Python programming language. In addition, all parts of the GUI are implemented using the PySide (QT) module [45].

5.1.1 Real-time Mode

The *real-time mode* functionality is implemented in the `BagRecorder` class. This is the part of the Process Visualizer that controls the recording of data, as well as the real-time mode specific parts of the GUI. Through this class, data on topics can be recorded as *.bag* files, based on the options specified in the GUI. The actual recording is done by the *Rosbag* command line tool [32]. Which topics to record is specified by the selected devices and their associated topics, defined in the configuration file. Figure 5.3(a) shows what the GUI looks like when the program is in standard real-time mode. In Figure 5.3(b), the program is also recording data.



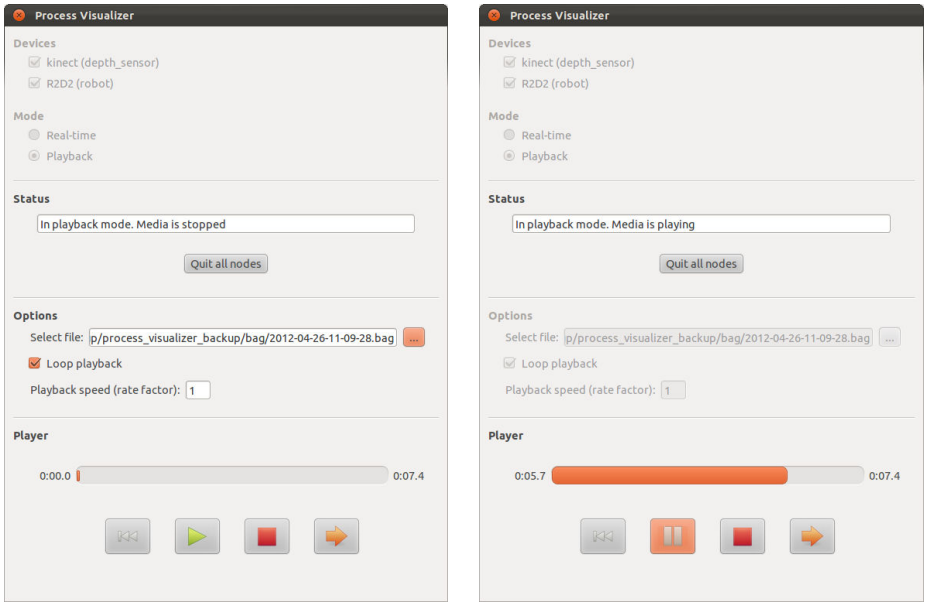
(a) Real-time mode. (b) Currently recording data.

Figure 5.3: Launcher GUI, in real-time mode.

The only possible recording options in the current implementation, is to specify what file name to use. This can be the current date and time, either with or without a prefix, or a file name specified by the operator.

5.1.2 Playback Mode

In a similar way, the *playback mode* functionality is implemented in the `BagPlayer` class. Here, it is possible to visualize a process by playing back data from a *.bag* file, again by the use of *Rosbag*. The `BagPlayer` class updates the playback functionality of the GUI, and is controlling the actual playback of data. Screenshots of the GUI is shown in Figure 5.4. Here, 5.4(a) shows the program in standard playback mode, while 5.4(b) shows the program while actually playing back data.



(a) Playback mode. (b) Currently playing back data.

Figure 5.4: Launcher GUI, in playback mode.

As can be seen in Figure 5.4, there are some different options in playback mode. First of all, it is possible to select which *.bag* file to use. The default location to search for these files are set to the `/bag` directory in the `process_visualizer` package. When a new *.bag* file is selected, information about this file is displayed in the terminal window. For the whole system to work, the selected file must contain a recorded version of the topics used by the currently selected devices.

Other possible options in this mode are selecting whether or not to loop the playback, and to set the playback speed. If the *loop playback* box is checked, the

playback will simply restart when it reaches the end. The other option is used to specify the playback speed, or rate factor, of the playback. For instance, a rate factor of 2, means to play at double speed – twice as fast as it was recorded.

The playback mode also includes a media player, to control the playback. This includes a progress bar and four buttons. The progress bar indicates how far into the playback the system is, and is shown as 'm:s.ms' (minutes, seconds, milliseconds). This information is continuously sent to this node from Rosbag. Furthermore, there are four buttons that can be used to control the playback. The first of these stops the playback, and goes to the beginning of the *.bag* file. The second button is the *play/pause* button. By pressing *play*, the selected *.bag* file starts playing with the selected options. In practice, this means that Rosbag is started – with some specific arguments. Then, by pressing *pause*, the system will pause the playback. When this happens, the GUI node sends a pause signal to Rosbag. This signal is also sent to resume the playback. Moreover, the *stop* button is used to stop the playback. What really happens here is that the GUI node tells Rosbag to exit. When the playback has stopped, the user is able to choose some new playback options, such as selecting a different file to play from. Finally, the last media button is *step*. This is only possible if the player is paused, and also works by sending a signal to Rosbag. Here, a *step* means to publish one single message. For instance, if a *.bag* file contains 10 different topics, only one of these will receive a new message for each step.

5.1.3 RViz Configuration

In order to be able to select which devices to display in RViz, a configuration file has to be created dynamically. This happens when the user presses the *launch nodes* button in the Launcher GUI. Then, based on which nodes are selected, a *.vcg* (RViz) configuration file is generated. Which display types to include in RViz is specified either by the device type, or explicitly in the XML configuration, by using the `<rviz>` tag (see Section 5.2). Currently, the supported display types are: *'RobotModel'*, *'MarkerArray'*, *'Marker'* and *'PointCloud2'*. To add support for more display types, the `RVizConfig` class (defined as a part of the

`process_visualizer` package) must be edited. Both *'RobotModel'* and *'MarkerArray'* are used by default for each robot device, and *'PointCloud2'* is used for a depth sensor. More details about the different display types can be found in Section 3.1.1.

Creating the RViz configuration file is implemented as a Python class called `RVizConfig`. This configuration file is created by first adding a default part, with standard configurations, such as setting the background color and the fixed frame, as well as adding a *'Grid'* to the scene. The rest is then created dynamically by the GUI node, based on the selected devices.

To update its appearance, each of the display types requires a topic, on which to receive new data. This is set to `'name'/robot_description` for a *'RobotModel'*, `'name'/marker_array` for a *'MarkerArray'*, `'name'/marker` for a *'Marker'* and `'name'/points` for a *'PointCloud2'*. For all of these, `'name'` refers to the name specified for a `<device>` tag in the XML configuration.

5.2 XML Configuration

The XML configuration is an important part of the implementation, in order to have a system that supports different kinds of devices. This can be used to specify all devices available in the system, as well as a lot of details about these devices. The devices can be a robot, a depth sensor, or some other device. For all these devices, it is possible to specify the position/orientation of the base frame, which extra nodes or launch files to start together with it, and which ROS topics the device uses. In addition, for robots, the URDF / Xacro model should also be specified here. Furthermore, if the device uses some special RViz display types (that is, not standard, such as *'PointCloud2'* for depth sensors), this can also be specified in the XML configuration.

To use this XML configuration, a Python class called `XmlParser` has been created. As can be seen in Figure 5.1, this class is used by the Launcher GUI, as well as the Robot/Depth Data Processor nodes. To summarize, it is used for:

- Creating the list of selectable devices in the Launcher GUI
- Specifying the device base frame, used to launch a ROS node to calculate/publish the transform (`static_transform_publisher`)
- Dynamically generating the RViz configuration file
- Finding which topics to record
- Indicating which topics contain point clouds, force sensor data, tool center frame, etc.
- Specifying the Xacro / URDF model that describes the robot device
- Listing all extra nodes / launch files that is needed for a device

All supported XML tags and a description of these can be found in Section 5.2.1, followed by a short example in Section 5.2.2 to show how it can be used. In addition to this, Chapter 6 presents a specific application, in which a more complex XML configuration is used.

5.2.1 Supported XML Tags

<**visualizer**>**:** Simply the root tag.

Elements: `<device>`

<**device**>**:** Main tag for describing a device.

Attributes:

`type` (required)

The device type. Supported types are: *'robot'*, *'depth_sensor'* and *'other'*. This information is used to decide which nodes to launch by the Process Visualizer.

`name` (required)

The device name (should not contain a space). E.g. *'my_robot'*

Elements: `<base>`, `<model>`, `<node>`, `<launch>`, `<topic>`, `<rviz>`, `<force>`.

<**base**>**:** The base frame, position and rotation of the devices' origin. A `<device>`

tag may only contain one of this tag.
Attributes:

`frame` (required)
: The base frame of the device, relative to the global frame

`xyz` (optional – defaults to *'0 0 0'*)
: The position of the base frame, relative to the global frame

`rpy` (optional – defaults to *'0 0 0'*)
: The rotation of the base frame, yaw/pitch/roll (in radians). Represented as an intrinsic rotation: first yaw (around z), then pitch (around y) and finally roll (around x).

<**model**>**:** Used to describe the robot model. This only applies to robots, and the file must be either URDF or Xacro (file name ending in *.urdf* or *.xacro*). A `<device>` tag may only contain one of this tag.
Attributes:

`pkg` (required)
: ROS package, in which the robot model is stored

`path` (required)
: Exact path for the model file, inside the package described above

<**node**>**:** This tag is used to specify ROS nodes to launch together with the device. Similar to the Roslaunch XML format, with the addition of the *ifstate* element. A `<device>` tag may contain several elements of this tag.
Attributes:

`ifstate` (optional – defaults to all states)
: By setting this attribute, the node will only be launched if the system is in the specified state (mode). Possible values are *'realtime'* and *'playback'*.

`pkg` (required)
: ROS package, in which the node exists

`type` (required)
: Node type – the executable file

`name` (required)
: Node name (cannot contain a namespace)

`ns` (optional – defaults to the device name)
: Namespace to start the node in

`args` (optional)
: Arguments to pass to the node executable

`output` (optional – defaults to *'screen'*)
: Specifies whether to send stdout to screen (*'screen'*) or a log file (*'log'*)

<**launch**>**:** This tag is used to specify ROS launch files (*.launch*) to launch together with the device. A `<device>` tag may contain several elements of this tag.

Attributes:

`ifstate` (optional – defaults to all states)
: By setting this attribute, the file will only be launched if the system is in the specified state (mode). Possible values are *'realtime'* and *'playback'*.

`pkg` (required)
: ROS package, in which the launch file exists

`type` (required)
: Name of the launch file

<**topic**>**:** Topics used by a device. A `<device>` tag may contain several elements of this tag.

Attributes:

`name` (required)
: The topic name (which cannot contain a namespace)

`ns` (optional – defaults to *''*)
: Namespace of the topic.

`type` (optional)
: What type of topic this is (what its function is). This information is used by *robot_proc* and *depth_proc*, to know which topics to listen to for messages. If this attribute is excluded, the topic will still be recorded/played back, but not used directly by any of the data processing nodes. Possible values are *'joint_states'*,

'tool_center_frame', *'force_sensor'* and *'pointcloud'*

`record` (optional – defaults to True)

Whether or not to actually record this topic. In a standard configuration, this is used for the point cloud topic, which is used by *depth_proc*, but is not recorded. Instead, the topics providing RGB/depth images are recorded (to use less hard disk space). Possible values are *'True'* and *'False'*

<**rviz**>**:** RViz display types associated with a device. This is mostly useful for *'other'* device types, as display types for robots and depth sensors are added automatically. A `<device>` tag may contain several elements of this tag.

Attributes:

`type` (required)

The RViz display type. Currently, the supported types are: *'RobotModel'*, *'MarkerArray'*, *'Marker'* and *'PointCloud2'*. To add support for more display types, the `RVizConfig` class (defined as a part of the `robot_visualizer` package) must be edited.

<**force**>**:** This tag is used to specify the force sensor frame. It was added to make it possible for the force sensor have a different frame than the robot tool.

Attributes:

`parentframe` (required)

The parent frame name (typically the tool frame)

`frame` (required)

Name of the force sensor frame

`xyz` (required)

Position of the frame, relative to the parent frame

`rpy` (required)

Rotation of the frame, relative to the parent frame. Rotation is defined in the same way as for the `<base>` tag

5.2.2 Example Usage

```
<?xml version="1.0" encoding="utf-8"?>
<visualizer>
    <device type="depth_sensor" name="kinect">
        <base frame="/camera_link" xyz="-1 0 1.5" rpy="0 0.1 0" />
        <topic ns="camera/depth_registered" name="camera_info" />
        <topic ns="camera/depth_registered" name="image_raw" />
        <topic ns="camera/driver" name="parameter_descriptions" />
        <topic ns="camera/driver" name="parameter_updates" />
        <topic ns="camera/rgb" name="camera_info" />
        <topic ns="camera/rgb" name="image_color" />
        <topic ns="camera/depth_registered" name="points"
               type="pointcloud" record="False" />
        <launch ifstate="realtime" pkg="sewing_cell"
                type="openni_live.launch" />
        <launch ifstate="playback" pkg="sewing_cell"
                type="openni_play.launch" />
    </device>
    <device type="robot" name="R2D2">
        <base frame="/R2D2/base_link" xyz="2 0 0.5"
              rpy="1.570796 0 0" />
        <model pkg="r2d2" path="urdf/R2D2.xacro" />
        <node pkg="r2d2" type="r2d2_transformer.py"
              name="r2d2_transformer" />
    </device>
</visualizer>
```

Listing 5.1: Example XML configuration file. This is the configuration file for the Launcher GUI and RViz view shown in Figure 5.2.

Listing 5.1 shows an example configuration, using a depth sensor (Kinect) and an imaginary robot (R2D2). The corresponding Launcher GUI and RViz view is shown in Figure 5.2. In this example, the Kinect is used as depth sensor, and for this device the XML configuration contains details about the camera base frame, all 7 topics it uses, and its associated launch files. The `<base>` tag says that the camera's base frame is called `/camera_link`, and is positioned $(x,\ y,\ z)^T = (\text{-}1,\ 0,\ 1.5)^T$, with a rotation $(\psi,\ \theta,\ \phi)^T = (0,\ 0.1,\ 0)^T$, relative to the global frame. As explained in the previous section, this rotation is given as

Euler angles yaw/pitch/roll. Moreover, for this device's topics, only the first 6 will actually be recorded by Rosbag. As can be seen in the listing, the last topic, `camera/depth_registered`, has the *record* attribute set to *False*, and will thus not be recorded. The reason for this is that this is a point cloud, which takes up a large amount of hard disk space. The large amount of hard disk usage for point clouds is also the reason why there are two different launch files associated with the Kinect sensor – one for real-time mode, and one for playback mode. In real-time mode `openni_live.launch` will be launched, and in playback mode `openni_play.launch`. Both of these launch all camera nodes necessary for processing the image data and creating point clouds. However, unlike the second, the first also launches the Kinect drivers, used to actually retrieve sensor data from the device. In theory, a different approach could be to only include the `<topic>` tag describing the point cloud. However, this would result in a much larger *.bag* file.

There are three tags describing the robot in this example. The `<base>` works exactly as for the Kinect. In addition, it also has a `<model>` tag. This tag describes the path of the Xacro (or URDF) file which describes the robot. In this case this file is in a ROS package called `r2d2`. This is also the location for `r2d2_transformer.py` node, which is used to do some processing specifically for this robot.

5.3 Robot Data Processor

Robot Data Processor is the part of the system that processes data from robots. It is implemented as a Python class called `RobotProcessor` in the `robot_proc` module, and is run as a separate node for each robot in the system. Mainly what this node does, is to receive sensor data from a robot's drivers/middleware, process this information, and send the processed data to the visualization node. More specifically, this node subscribes to topics published by the robot middleware, to receive the current robot joint states, tool center frame and force sensor readings, which again is used for updating the robot model pose and visualizing the tool velocity and force measurement.

First of all, this node uses the `XmlParser` to get information about the robot from

the XML configuration file. That is, information about the robot model, as well as the topics names, on which the different sensor data is available. The robot model can be of either URDF or Xacro format, and its path is specified in the `<model>` tag in the XML configuration. This model is then parsed, and set as a parameter with the name `'name'/robot_description`. Furthermore, when the parameter is set, the *Robot State Publisher* node is launched. The main reason for launching this node from the Robot Data Processor node is because it requires the `robot_description` parameter to be set.

The final initialization done by the Robot Data Processor, is to fetch the topics for the joint states, tool center frame and force sensor data. This is specified as the `type` attribute in a `<topic>` tag. To successfully publish (and visualize) the joint states, velocity vector and force vector, the corresponding topics must be specified in the XML configuration. This node will process and publish the available data. For instance, if the tool center frame topic is not specified, the velocity vector will be ignored, and not visualized. Moreover, this node publishes new information as soon as new data is received and processed. This applies to the joint states, velocity vector and force vector. To provide the user with extra information, a textual description of the actual values of the velocity and force is generated and published together with the arrow markers. All of these markers are then visualized by RViz.

5.3.1 Joint States

The joint states are received as a `std_msgs/Float32MultiArray` data type, on the topic defined in the XML configuration. This data is then transferred into a `sensor_msgs/JointState` data type. The reason for this is that the Robot State Publisher requires `JointState` messages. This data type basically consists of an array with the joint names and an array with the current positions/angles. The actual calculation of the different link frame transformations is done in the Robot State Publisher node, based on the current value of the states.

5.3.2 Velocity Vector

The velocity vector is generated based on the change in the position of the tool center frame. More specifically, it is given by

$$\mathbf{v}_{t,k} = \frac{\mathbf{p}_{t,k} - \mathbf{p}_{t,k-1}}{(t_k - t_{k-1})r} \tag{5.1}$$

where $\mathbf{v}_{t,k}$ is the velocity, $\mathbf{p}_{t,k}$ is the position, t_k is the time, and r is the *rate*. Because the tool center frame messages that are received from the robot drivers do not contain a timestamp, the time used in (5.1), t_k, is the time when the specific message is received. Thus, the rate has to be included in this calculation. In real-time mode the rate is always 1, and in playback mode it depends on the playback rate set in the Launcher GUI.

Furthermore, the arrow marker used to indicate the tool velocity is defined in the robot's base frame. Thus, to make the arrow appear at the location of the tool, its position it set to the position values (first three array values) given by the tool center frame message. The orientation of an arrow marker is defined such that the arrow points in the direction of the local x-axis. The axis-angle representation of this rotation is then given by

$$\mathbf{k} = \cos^{-1} \frac{\mathbf{v}_a \cdot \mathbf{v}_t}{\|\mathbf{v}_a\|\|\mathbf{v}_t\|} \tag{5.2}$$

$$\theta = \frac{\mathbf{v}_a \times \mathbf{v}_t}{\|\mathbf{v}_a \times \mathbf{v}_t\|} \tag{5.3}$$

where $\mathbf{v}_a = (1, 0, 0)^T$ and $\mathbf{v}_t$ is the current tool velocity. The reason for this $\mathbf{v}_a$ is because the necessary rotation is from the base frame to the tool frame, such that the new x-axis is pointing in the direction of the velocity vector. Converting this representation into quaternion is done by the following formulas [46]:

$$q_x = k_x \sin\frac{\theta}{2} \tag{5.4}$$

$$q_y = k_y \sin\frac{\theta}{2} \tag{5.5}$$

$$q_z = k_z \sin\frac{\theta}{2} \tag{5.6}$$

$$q_w = \cos\frac{\theta}{2} \tag{5.7}$$

Finally, the length of the arrow marker is scaled linearly based on the magnitude of the velocity vector. To make sure the arrows are visualized informatively, an upper and lower limit for the arrow length are used. That is, if the velocity is below the lower limit, the arrow length will still be set to the minimum value, and similarly for the upper limit.

5.3.3 Force Vector

The values of the force vector is fetched from the first three values of the force sensor topic, as x, y and z, respectively. The frame used by the arrow marker representing the force is specified in the XML configuration. If there is no force frame specified there, the tool frame will be used. The force vector is rotated similarly to the velocity vector, by using $\mathbf{v}_a = (1, 0, 0)^T$ and $\mathbf{v}_t = \mathbf{v}_f$, where $\mathbf{v}_f$ is assumed to be the measured force in the force frame. Thus, to make the arrow point in the direction of the applied force, the orientation is calculated using Equation (5.2) - (5.7). As this marker uses the force frame, its position is set to the origin.

The length of the force marker is set similarly to the velocity marker, by linearly scaling the magnitude of the vector, with an upper and lower limit.

5.4 Depth Data Processor

Depth Data Processor is a node created to process data from depth sensors. It is implemented as a Python class called `DepthProcessor`, in the `depth_proc` module. In the implementation done for this thesis, this node is very simple, because of the very little general depth data processing necessary. At initialization, the point cloud topic is fetched from the XML configuration. This is the `<topic>` tag, for

which the `type` attribute is set to *'pointcloud'*. Then, for all messages received on this topic, the timestamp is updated. Finally, the altered point cloud is then published to the `'name'/points` topic. Both the incoming and outgoing messages must be of the data type called `sensor_msgs/PointCloud2`.

The purpose of this node is to make the point cloud data be accepted by RViz. For instance, the transformations sent by the Robot State Publisher have the current time as timestamp. Thus, if the point clouds have an old timestamp, such as in playback mode, RViz will not accept the messages.

5.5 Summary

The general system that is implemented for this thesis, includes a GUI, a Robot Data Processor, and a Depth Data Processor node. The GUI is used to select which devices should be visualized, as well as to record and play back data. Furthermore, the data processor nodes do the processing that is common for all robots and for all depth sensors. To make this system dynamic, an XML configuration is used to specify information about all available devices in the system.

Adding a new device to the system is fairly easy. Assuming the device drivers are available, all that has to be done is adding the device to the XML configuration. This includes the path to the model description (URDF/Xacro) if it is a robot, and possibly any additional nodes/launch files, that are used by the corresponding device. Additionally, if the device for some reason requires an RViz display type that is not described in Section 5.1.3, this has to be added manually, by altering the `RVizConfig` class.

Chapter 6

Application: Sewing Cell

To be able to utilize the system described in Chapter 4 and 5, it must be customized for a specific case, or application. In this chapter a real application, for which this 3D visualization software is useful, is explained. This application is the Ekornes sewing cell at SINTEF Raufoss Manufacturing. The sewing cell originally consists of two industrial robots and a sewing machine, where the two industrial robots are used to feed the sewing machine with leather covers (for sofas) to sew together. In addition to this, a Kinect sensor is used together with this application, to create a better view of the complete scene. The Kinect is mounted above the rest of the devices, to visualize the workpieces in the best possible way. A more detailed description of the sewing cell can be found in [47].

This chapter presents the program specification for the sewing cell application in Section 6.1. This is the specification given by the researchers working on this application at SINTEF Raufoss Manufacturing. Furthermore, Section 6.2 presents the setup used to customize the system for this specific application. This includes both the laboratory setup and the software adjustments. Finally, Section 6.4 presents the results from testing the complete system on the sewing cell application.

6.1 Specification

To generate a complete picture of the Ekornes sewing cell at SINTEF, the program should contain the following:

- **Two modeled robots (Universal Robots UR-6-85-5-A):** These robots should be modeled using URDF, and their motion should be visualized by receiving measurement data on ROS topics.
- **A static model of a sewing machine**
- **Measurements** (visualized as vectors/arrows, available via ROS topics) **of:**
  - Robot tool velocity
  - Robot tool force sensor
  - Sewing machine feeding speed
  - Sewing machine edge sensors
- **3D sensor data from the Kinect:** Used to provide a clear picture of the parts of the scene that are not modeled. This data is received via ROS topics.

This should be implemented in ROS, utilizing RViz as 3D visualization software. In addition, the program should be able record the measurement data, to make it possible to visualize the process later by playing back the data. The recording / playback of data should be controlled through the use of a simple GUI.

6.2 Setup

Figure 6.1 shows an expanded version of the system overview presented in Chapter 4. In this figure, all the devices used in the sewing process are included: Two industrial robots (Universal Robots UR-6-85-5-A), a Kinect sensor and a sewing machine. Each of the objects inside the main circle are truly the same as in Figure 4.1. However, since there are two industrial robots, two *Robot Data Processor* nodes are used. Additionally, the *Robot State Publisher* node is excluded from the figure to create a more orderly picture. For all devices, the device specific data processing is happening inside the dashed line objects. This device specific data processing is described in more detail in Section 6.2.

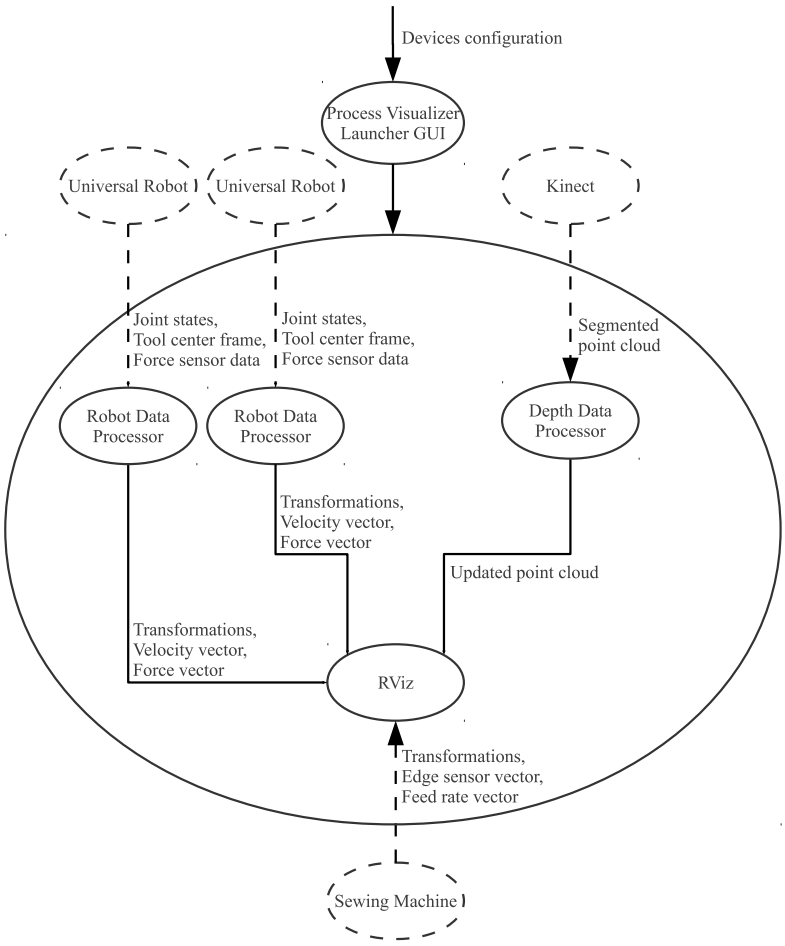


Figure 6.1: An overall overview of the sewing cell.

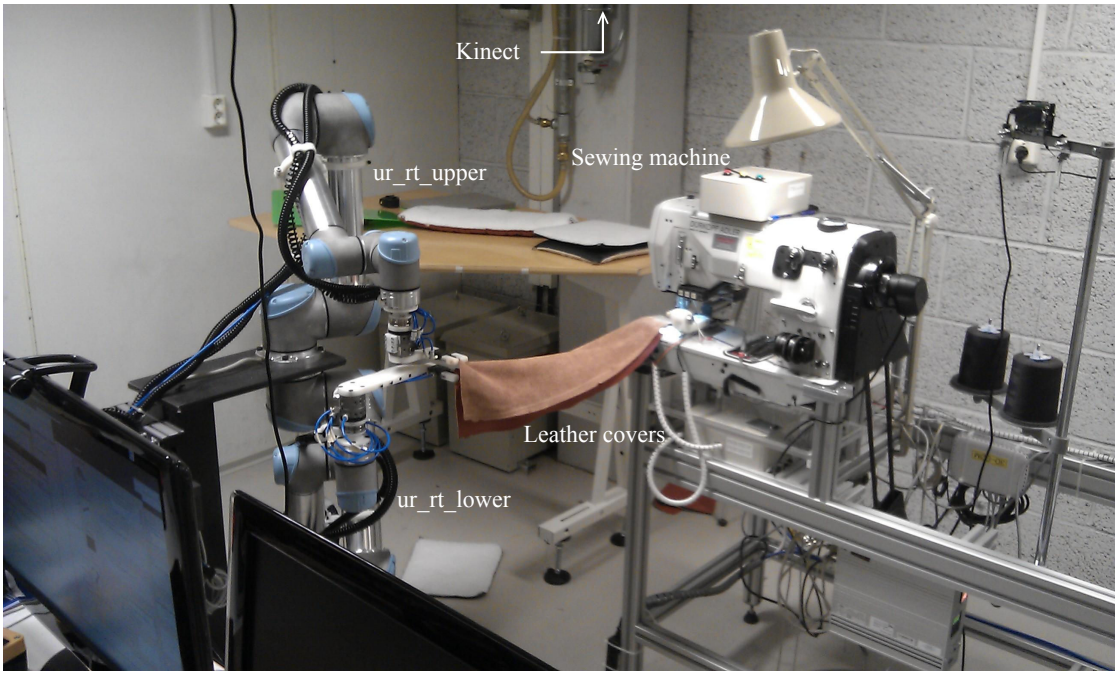


Figure 6.2: Laboratory setup for the sewing cell.

The laboratory setup can be seen in Figure 6.2. Here, the two robots `ur_rt_upper` and `ur_rt_lower`, the sewing machine and the leather covers can be seen, as well as an indication to where the Kinect sensor is placed. The XML configuration used to visualize this process is presented in Section 6.2.4. A different image of the sewing cell is also included in Figure 6.3. This figure shows the process from a slightly different angle, with the computer monitor included. On this monitor, the RViz view of the process in real-time can be seen.

6.2.1 Industrial Robots

The robots used in the sewing cell are two Universal Robots [48] UR-6-85-5-A 6-axis industrial manipulators [47]. These two robots are used to hold and control the leather covers, in order to sew them precisely together. The drivers and ROS middleware for the robots are developed at NTNU / SINTEF Raufoss Manufacturing. The visualization software described here assumes joint states, force sensor data, tool center frame are available on topics. This way, no extra data processing

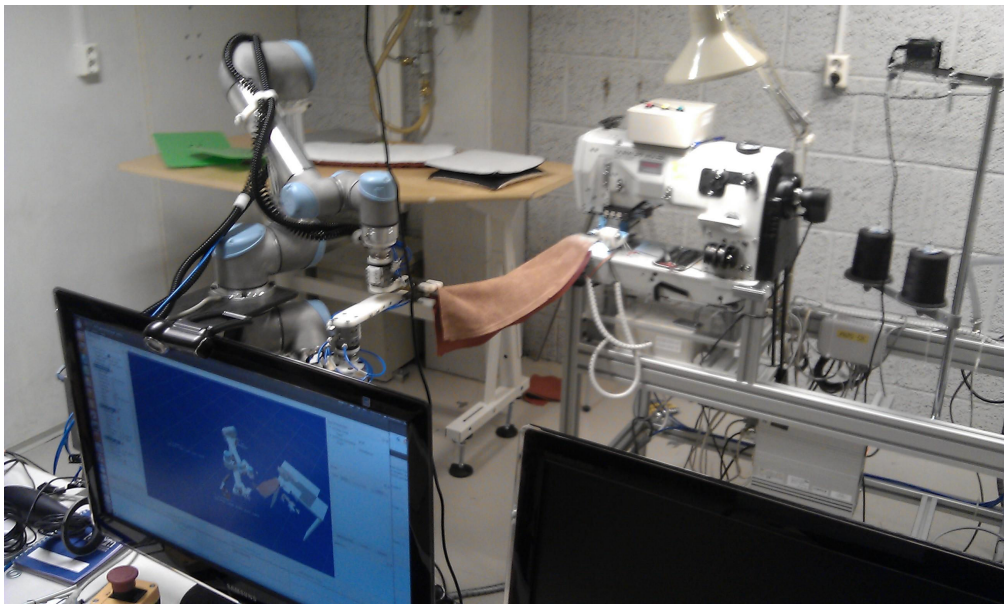


Figure 6.3: Laboratory setup for the sewing cell. Here, the computer monitor showing the visualization software can also be seen.

nodes, except for the Robot Data Processor (Section 5.3), are needed. Table 6.1 lists the DH parameters describing the robots. In addition, the home position is given by

$$\mathbf{q}_{home\_offset} = \left(0,\ 0,\ -\frac{\pi}{2},\ 0,\ -\frac{\pi}{2},\ 0,\ 0\right)^T \tag{6.1}$$

To create the model of the industrial robots, the DH parameters in Table 6.1 are used. This model is the one used for visualizing the robots in RViz, and is created using URDF and Xacro. In Appendix D, the exact Xacro expressions are shown. This *.xacro* file is translated into a URDF file during runtime. Translating Xacro into URDF is done by the *Xacro* script [36]. A screenshot of the robot, created using Xacro/URDF, is shown in Figure 6.4.

In Figure 6.4, it can be seen that the tool used to grasp the workpieces is not included, and that the colors are a bit off compared to the real robot. The reason for this is that the model is based on existing parts, and perfecting these to have an exact copy of the robot was not prioritized, since the current tool is just a prototype and will be replaced in the near future.

Link	$\mathbf{a}_i$	$\boldsymbol{\alpha}_i$	$\mathbf{d}_i$	$\boldsymbol{\theta}_i$
0	0	0	0	θ_0^*
1	0	$\pi/2$	0.0892	θ_1^*
2	-0.425	0	0	θ_2^*
3	-0.39243	0	0	θ_3^*
4	0	$\pi/2$	0.109	θ_4^*
5	0	$-\pi/2$	0.093	θ_5^*
6	0	0	0.082	θ_6^*

Table 6.1: DH parameters for the Universal Robots UR-6-85-5-A.

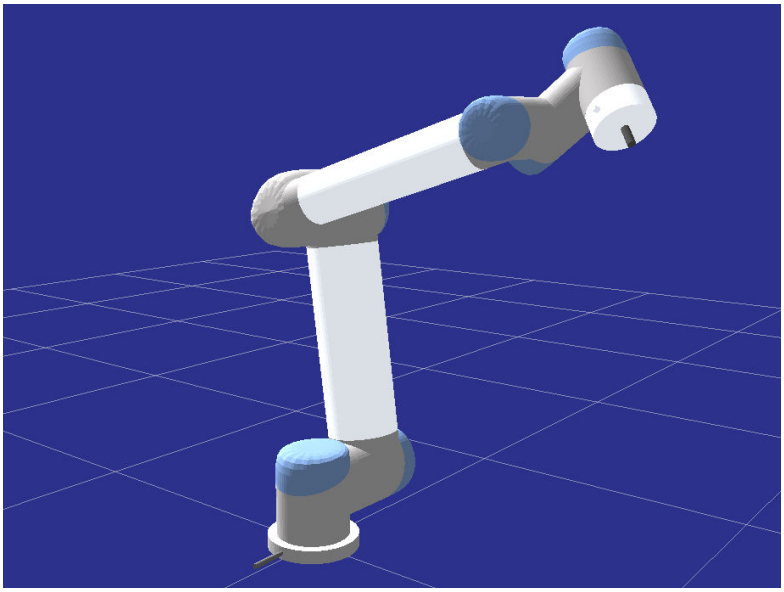
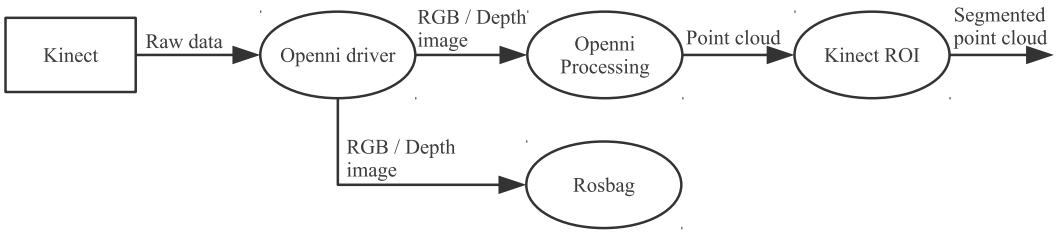


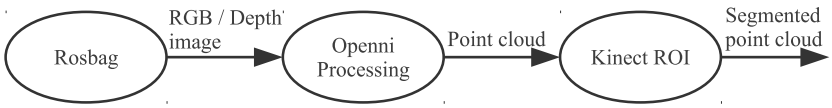
Figure 6.4: Screenshot of an industrial robot, visualized in RViz.

6.2.2 Kinect

The depth sensor used for the sewing cell is a Kinect. This sensor is mounted near the ceiling, pointing down at the rest of the scene. The reason for choosing this location is to provide an image of the workpieces in the best possible way. All of the processing specific for the Kinect (not depth sensors in general), is done inside the dashed lined object called *Kinect* in Figure 6.1. A more detailed view of this object's components is shown in Figure 6.5. Figure 6.5(a) shows the components in real-time mode, and Figure 6.5(b) shows the components in playback mode.



(a) System components for the Kinect in real-time mode.



(b) System components for the Kinect in playback mode.

Figure 6.5: Kinect components.

In this figure, the rectangular object illustrates the actual hardware of the Kinect. To communicate with the hardware, the OpenNI drivers are used (see Section 3.3.2 for further information). These drivers retrieves an RGB and depth image from the Kinect. Further processing of this data is done in the node called *Openni Processing*. This is where the point cloud is generated, by combining the RGB and depth images. Both the OpenNI drivers and OpenNI Processing node are a part of the `openni_launch` ROS package. Furthermore, in real-time mode it is also possible to record data, using Rosbag. In playback mode, this data can then be played back. Again, the playback is done by Rosbag, which means that neither the actual hardware nor the OpenNI drivers are needed. The final data processing for the Kinect is done in the node called *Kinect ROI*. This node receives the point cloud, segments the region of interest (ROI), and sends the segmented point cloud

to the standard *Depth Data Processor*. The Kinect ROI node is created for this study, and is implemented in C++, due to the use of Point Cloud Library (PCL) [33]. Ideally, this node would be a part of the Depth Data Processor. However, this was not done because of the necessity for implementing the node in C++. Having this as a general component would require rewriting the Depth Data Processor, which again would require more complicated implementation, such as a C++ XML parser, a binding, or a call to the C++ program from Python.

The segmentation of the incoming point cloud is done based on the x, y and z coordinates of the points. That is, the inputted point cloud is filtered, such that the resulting point cloud only consists of the points within a given threshold. To make the point cloud fit nicely together with the rest of the scene in the visualization, the threshold values were chosen to be: $\Delta x = 0.5$ m, $\Delta y = 0.4$ m and $\Delta z = 0.5$ m. The criteria for whether to accept a given point $\mathbf{p} = (p_x,\ p_y,\ p_z)^T$ can then be described mathematically as

$$\|p_i - c_i\| < \Delta i, \quad \forall\ i = x, y, z \tag{6.2}$$

where $\mathbf{c} = (c_x,\ c_y,\ c_z)^T$ is the center of the segmented cube. The main goal of this segmentation is to get a resulting point cloud that only contains the points in the region of interest, which is the leather covers. Having a larger point cloud that covers the robots and sewing machine is unnecessary because of the static models, and appears more as noise than helpful information in the visualized scene.

6.2.3 Sewing Machine

For the general system described in Chapter 4, the sewing machine is considered as an *'other'* device. Thus, to process the sensor data from the sewing machine, a node called *Sewing Machine Data Processor* (`sewing_machine`) has been created. This node assumes that sensor data from the sewing machine is available on ROS topics, and uses this information to generate markers for the visualization. The sensors of interest for this application are two edge sensors, used to control the robots according to the current position of the two workpieces. That is, there is

one upper edge sensor, for the upper workpiece, and one lower edge sensor, for the lower workpiece. The upper robot then controls the upper workpiece, based on sensor data from the upper edge sensor and the force sensor attached to the robot tool, and similarly for the lower robot. A picture of the upper edge sensor can be seen in Figure 6.7. These edge sensors are illustrated in RViz as arrow markers, with length and direction indicating how far away from the setpoint the values are. The default value for the setpoint (for the visualization) is 255 (out of 512 possible data values for the sensor), but this can also be changed by setting the `/sewing_machine/setpoint_upper` (or 'lower') parameter. To affect the visualization, this parameter must be set before the visualization is launched from the Launcher GUI.

Figure 6.6: Close-up view of the sewing machine.

In addition to this, the data processor node also receives the current feed rate of the sewing machine. This information is used to generate a vector (arrow), which is used to illustrate the current feed rate, or estimate of the sewing machine velocity, in RViz. Furthermore, a textual description of all sensor data is also published by this node, to give a more detailed output of the actual values.

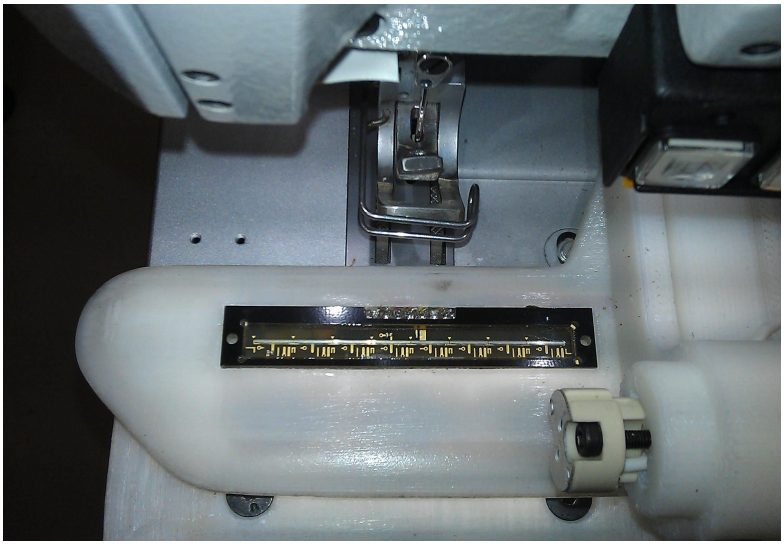


Figure 6.7: Detailed view of the upper edge sensor.

6.2.4 XML Configuration

Listing 6.1 shows the XML configuration used for the setup described in this chapter. By looking at this configuration, it can be seen that the two Universal Robots are called `ur_rt_upper` and `ur_rt_lower`. Both of these robots have three topics specified; one for the joint states, one for the tool center frame transformation and one for the force/torque sensor. In addition to this, they both also have a base frame and force frame specified, as well as the path to the Xacro file describing the robot model.

For the Kinect sensor, the XML configuration is almost exactly like the one presented in Section 5.2.2. The only difference is the addition of the *Kinect ROI* node, used to segment the point cloud. In addition to being specified as a `<node>` tag, an additional `<topic>` tag is also included. The reason for this is to specify the point cloud published by the Kinect ROI node as the point cloud to use in RViz (by setting the *type* attribute).

The sewing machine is of the device type *'other'*, which means it has no default nodes, unlike for robots and depth sensors. Thus, all additional nodes, topics and RViz display types must be specified in the XML configuration. As can

be seen below, the sewing machine depends on two nodes: `sewing_machine` and `static_transform_publisher`. The first of these is the node called *Sewing Machine Data Processor* (described in Section 6.2.3), and the second is a node used to publish static frame transformations (shortly presented in Section 5.1). Furthermore, the XML configuration also shows that the sewing machine subscribes to three topics and uses a *MarkerArray* RViz display type. All of these are used to calculate and visualize the markers/vectors described in Section 6.2.3.

```
<visualizer>
    <device type="robot" name="ur_rt_upper">
        <model pkg="sewing_cell"
               path="urdf/ur_rt_upper.urdf.xacro" />
        <base frame="/ur_rt_upper/link0" />
        <topic ns="ur_rt_upper" name="q_act"
               type="joint_states" />
        <topic ns="ur_rt_upper" name="tcf"
               type="tool_center_frame" />
        <topic ns="ft_upper" name="ft" type="force_sensor" />
        <force parentframe="/ur_rt_upper/tool"
               frame="/ur_rt_upper/force"
               xyz="0 0 0" rpy="-1.570796 0 0" />
    </device>

    <device type="robot" name="ur_rt_lower">
        <model pkg="sewing_cell"
               path="urdf/ur_rt_lower.urdf.xacro" />
        <base frame="/ur_rt_lower/link0" xyz="0 0 0"
              rpy="0 3.141592653589793 0" />
        <topic ns="ur_rt_lower" name="q_act"
               type="joint_states" />
        <topic ns="ur_rt_lower" name="tcf"
               type="tool_center_frame" />
        <topic ns="ft_lower" name="ft" type="force_sensor" />
        <force parentframe="/ur_rt_lower/tool"
               frame="/ur_rt_lower/force"
               xyz="0 0 0" rpy="-1.570796 0 0" />
    </device>

    <device type="depth_sensor" name="kinect">
```

```
        <base frame="/camera_link" xyz="-0.53 0.09 1.90"
              rpy="0 1.55 0" />
        <topic ns="camera/depth_registered" name="camera_info" />
        <topic ns="camera/depth_registered" name="image_raw" />
        <topic ns="camera/driver" name="parameter_descriptions" />
        <topic ns="camera/driver" name="parameter_updates" />
        <topic ns="camera/rgb" name="camera_info" />
        <topic ns="camera/rgb" name="image_color" />
        <topic ns="camera/depth_registered" name="points"
               record="False" />
        <topic ns="kinect" name="roi_points" type="pointcloud"
               record="False" />
        <launch ifstate="realtime" pkg="sewing_cell"
                type="openni_live.launch" />
        <launch ifstate="playback" pkg="sewing_cell"
                type="openni_play.launch" />
        <node pkg="sewing_cell" type="kinect_roi"
              name="kinect_roi" />
    </device>

    <device type="other" name="sewing_machine">
        <base frame="/sewing_machine/base"
              xyz="-0.667 -0.499 0.05" rpy="0 0 0" />
        <node pkg="sewing_cell" type="sewing_machine.py"
              name="sewing_machine" />
        <node pkg="tf" type="static_transform_publisher"
              name="needle_frame_transformer"
              args="-0.08 -0.08 0 1.570796 0 0 /sewing_machine/base
                    /sewing_machine/needle 100" />
        <rviz type="MarkerArray" />
        <topic name="edge_finder_upper" />
        <topic name="edge_finder_lower" />
        <topic name="feed_rate_est" />
    </device>
</visualizer>
```

Listing 6.1: XML configuration for the sewing cell.

6.3 Experiments

To test the implementation described above, a few different experiments will be done. First off all, the system will be tested by running the program in real-time mode, to see that everything works correctly. This testing includes making sure that the URDF models of the robots are visualized accordingly to the real robots. Moreover, there should also be a point cloud generated from the Kinect sensor, and different arrows indicating the tool velocity and force sensor measurements for each of the robots, as well as arrows for the sewing machine feeding speed and edge sensor measurements.

Furthermore, to make sure the system can be used for detailed analysis of a robotic process, the precision of the recording and playback will be tested, by running the program in real-time mode and record data on all topics. Then, this data will be played back, by running the program in playback mode. This testing will be done with the following configurations:

1. Using a Kinect, which publishes new data at a rate of 30 Hz
2. Using a Kinect, which publishes new data at a rate of 10 Hz
3. Not using a Kinect

The reason for doing these experiments is that the Kinect generates a lot of data that has to be recorded. For the Kinect sensor, a default publish rate is 30 Hz. How to set a different publish rate for the Kinect sensor is described in Appendix C. All the experiments described in this thesis are run on a standard desktop computer, running the Ubuntu operating system.

6.4 Results

By starting the program with the configuration specified in the previous section, the window in Figure 6.8 appears. This window is as presented in the earlier chapters, and includes a list of selectable devices, according to the given XML configuration.

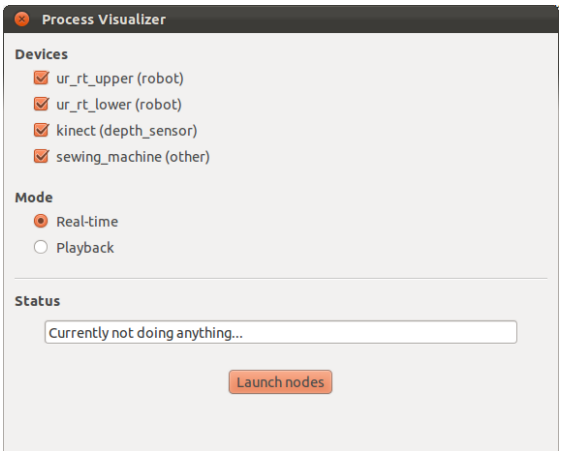
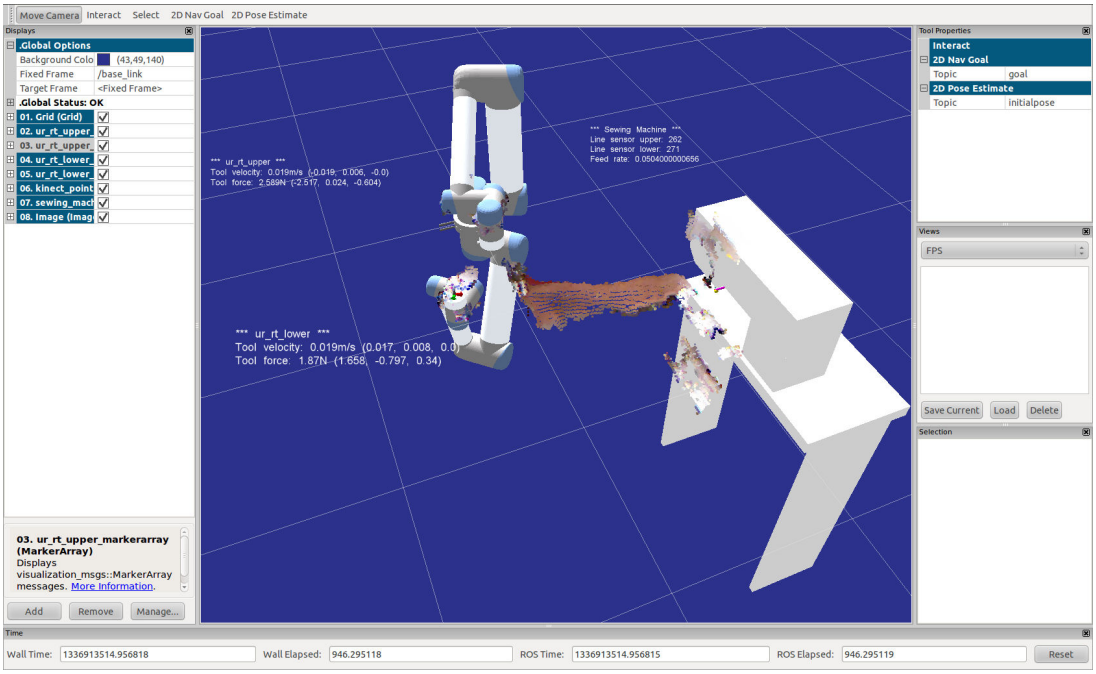


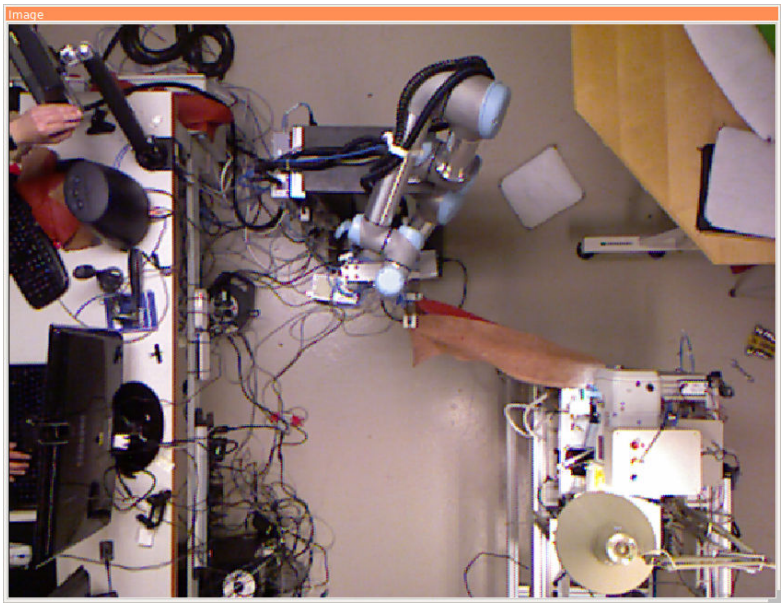
Figure 6.8: Launcher GUI for the sewing cell.

When launching the process with all devices selected from the window above, the RViz window in Figure 6.9(a) is shown. To show what the real process looks like, the RGB image from the Kinect is included in Figure 6.9(b). In Figure 6.9(a), the visualization of the complete sewing process can be seen, including everything mentioned in the specification in Section 6.1. That is, a sewing machine sewing together two leather covers, which are held (and controlled) by the two industrial robots.

First of all, this figure shows a model of both robots, the sewing machine and the 3D point cloud of the interesting parts of the process (the leather covers). Additionally, a textual description of all available sensor data is presented. This includes the tool velocity and force sensor data for both robots, as well as the sewing machine feed rate and sensor data readings for the edge sensors. Also, for each of the robots, there are two arrow markers; a green arrow used to indicate the end effector velocity, and a red arrow used to indicate the force applied to the tool. As mentioned earlier, the reason for the tool offset is because the robot model does not contain a model of the actual tool. Furthermore, for the sewing machine, there are three arrow markers; a purple arrow to indicate the feed rate, a yellow arrow to indicate the lower edge sensor, and a cyan (light blue) arrow to indicate the upper edge sensor. These arrow markers are shown in more detail in the following subsections.



(a) RViz 3D view.



(b) RGB image.

Figure 6.9: RViz 3D view and the corresponding RGB image of the complete sewing cell. The RGB image is included to see what the real process looks like.

Both in real-time and playback mode, it is possible to move around in the scene, cf. Figure 6.9(a), to view the process from all different angles. This makes it easier to look closely at specific parts of the process. Furthermore, in the menu on the left hand side it is also possible to dynamically select which devices to show. For instance, the point cloud can be removed, in order to see the other devices more clearly.

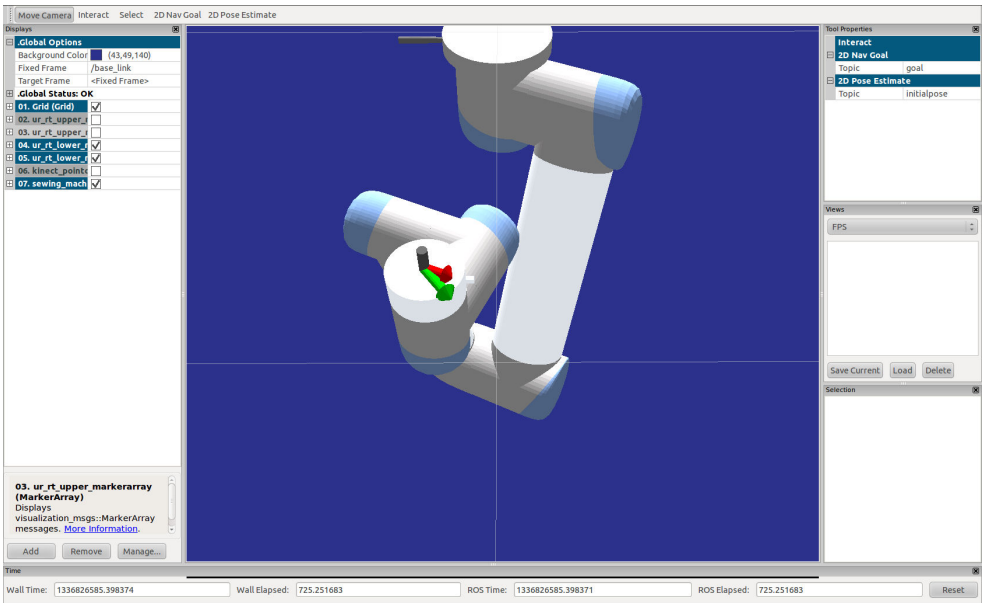


Figure 6.10: A closer view of the tool velocity and force sensor.

A closer look at the visualized vectors for the robots can be seen in Figure 6.10 - 6.12. The first of these screenshots shows the lower robot, and the corresponding velocity and force marker. In this view, both the upper robot and the point cloud are removed. Thus, a clear view of the lower robot can be seen. Here, the green arrow indicates that the end effector of the robot is moving to the bottom right, as seen in this image. This screenshot is taken from the same angle as Figure 6.9(a), which means that the robot is moving slightly right, towards the sewing machine. Similarly, the red arrow indicates a force applied in almost the same direction. The reason for this is that the sewing machine is currently sewing the workpieces together, which results in an applied force in the direction of the sewing. At the same time, the robot is moving slightly right, to feed the sewing machine correctly, by moving the (lower) workpiece in this direction.

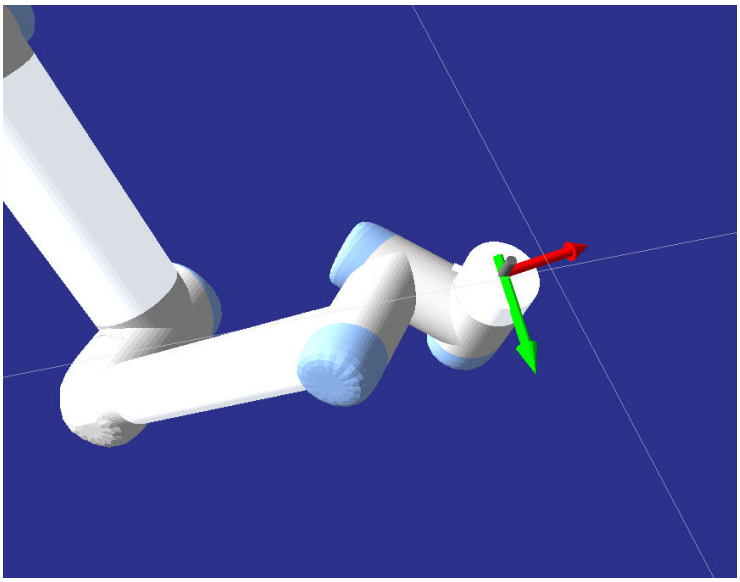
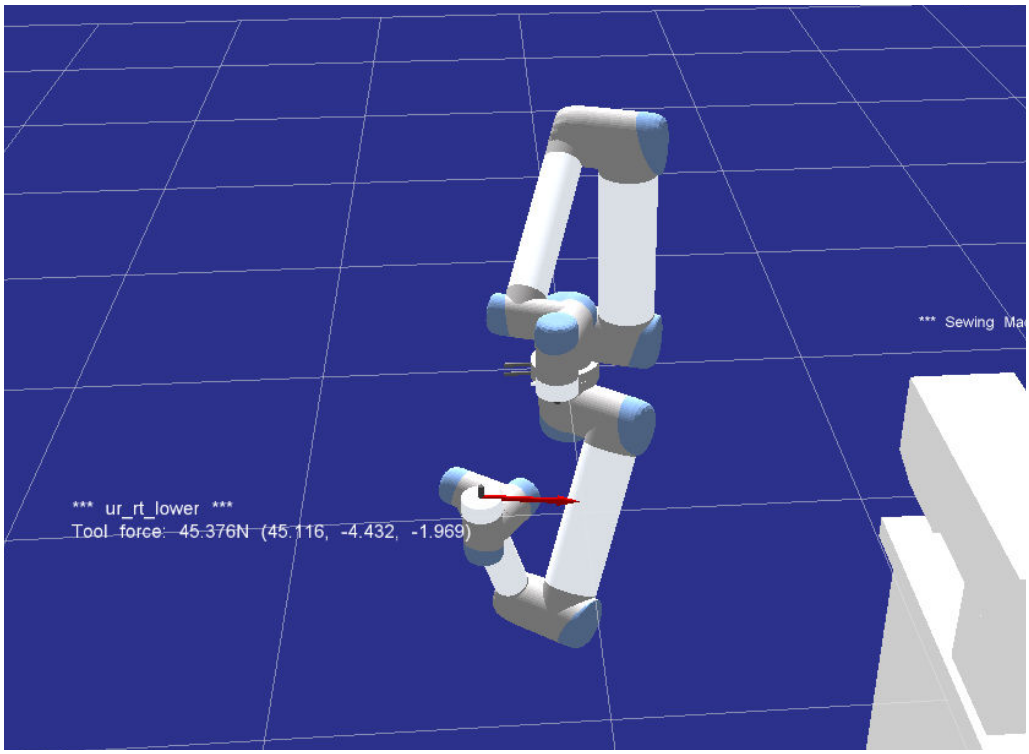


Figure 6.11: Close-up view of the lower robot, indicating a lagged velocity vector.

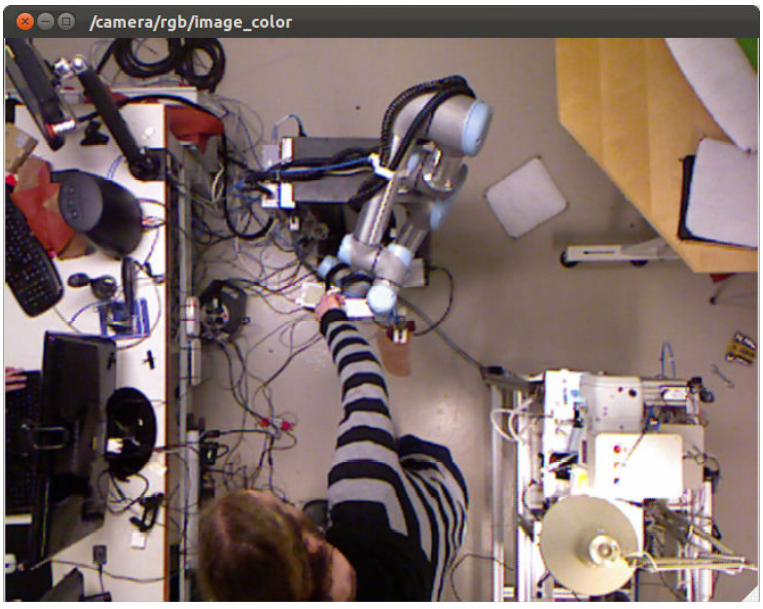
In Figure 6.11, a close-up view of the lower robot in shown. Here, the velocity arrow is slightly lagging, compared to the rest of the displays. Originally, the end of the arrow is supposed to be exactly at the tool center frame. However, because the different display data is published asynchronously by the *Robot Data Processor* node, the frames that the markers use are not always updated when drawn by RViz.

Figure 6.12 shows a human force applied to the robot tool. Here, the robot is not moving (at least not more than the lower threshold), which is the reason why there is no velocity vector. The force is also in this case applied in the direction of the sewing machine, and it has a magnitude of about 45 N, as can be seen in the textual description.

A close-up view of the sewing machine sensor data visualization can be seen in Figure 6.13. Here, the purple arrow indicates that the machine is currently running at the estimated feed rate given in the textual description (not shown in this figure). Furthermore, the yellow arrow illustrates that the lower workpiece is currently too far to the right. More specifically, the sensor reading from the edge sensor has a larger value than the setpoint, which means the corresponding workpiece should



(a) RViz 3D view.



(b) RGB image.

Figure 6.12: RViz view and the corresponding RGB image showing the force sensor. The RGB image is included to see what the real process looks like.

be moved left – towards the setpoint. Similarly, the visualization of the upper edge sensor is shown as a cyan (light blue) arrow, pointing left. This means that the upper workpiece is slightly to the left of where it should be. The corresponding values from the edge sensors are what is actually used by the robots to control the workpieces.

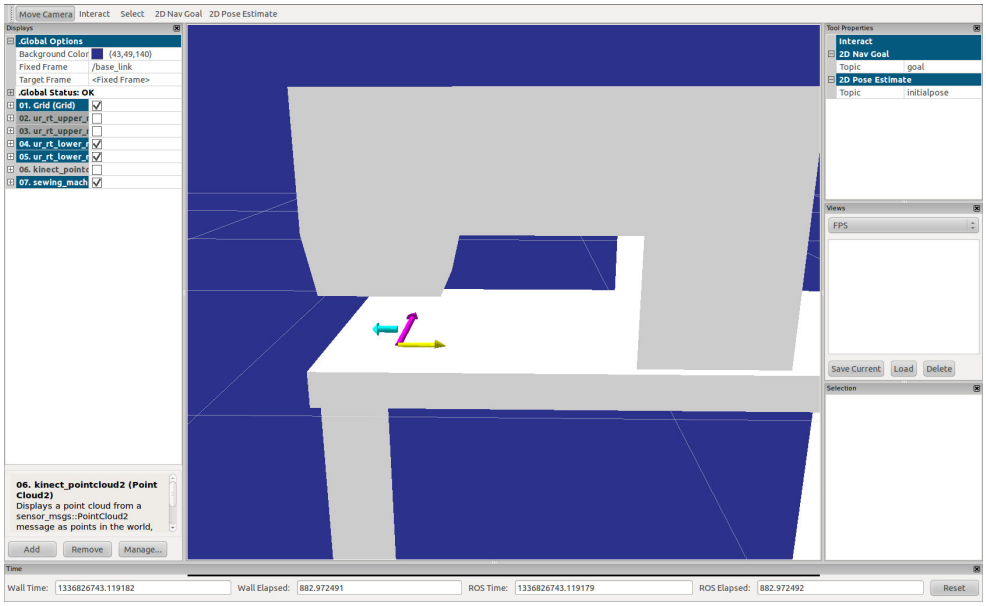


Figure 6.13: Close-up view of the sewing machine sensor data, visualized as arrow markers.

Appendix D lists some specifics about the sewing cell implementation. This includes the URDF/Xacro model of the Universal Robots, a list of all the (interesting) ROS nodes used in the sewing cell visualization and a figure showing all the different coordinate frames and their relationship.

6.4.1 Playback precision

This section presents the results from comparing the measurement data in real-time mode to the ones in playback mode. Firstly, Figure 6.14 shows the force sensor measurements in both real-time and playback mode. This figure illustrates that the played back data is very similar to the original data published by the drivers in real-time.

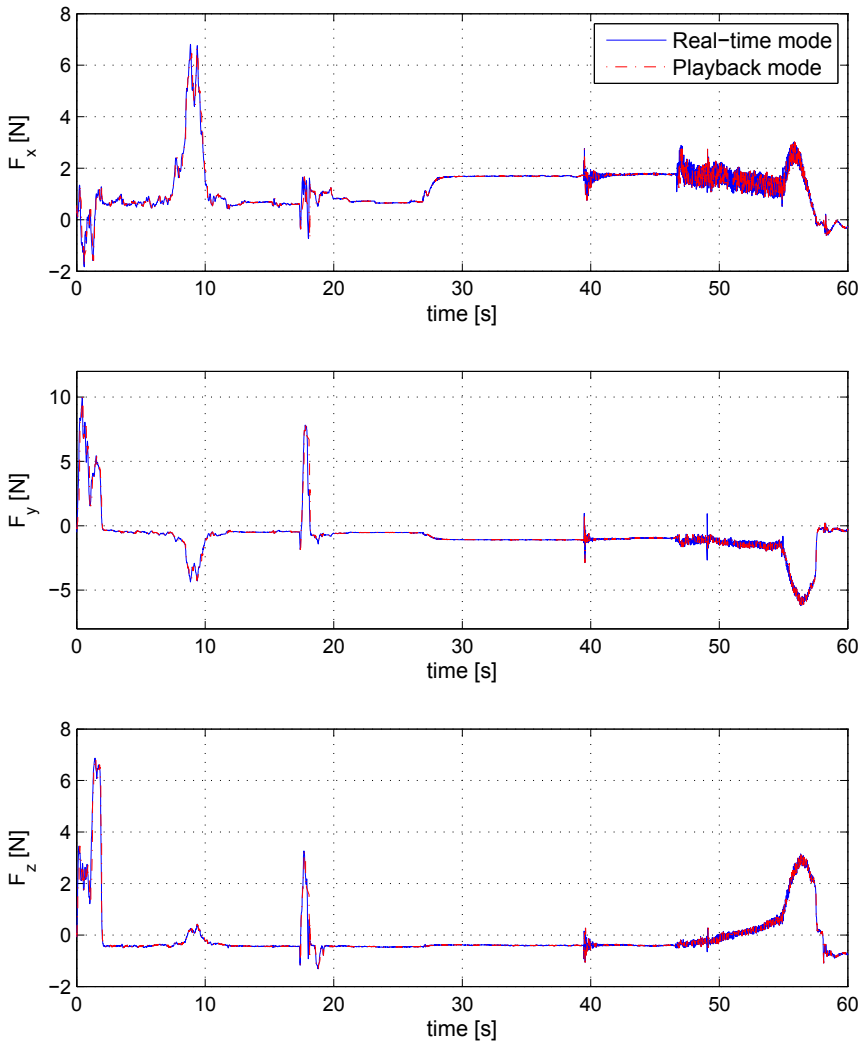
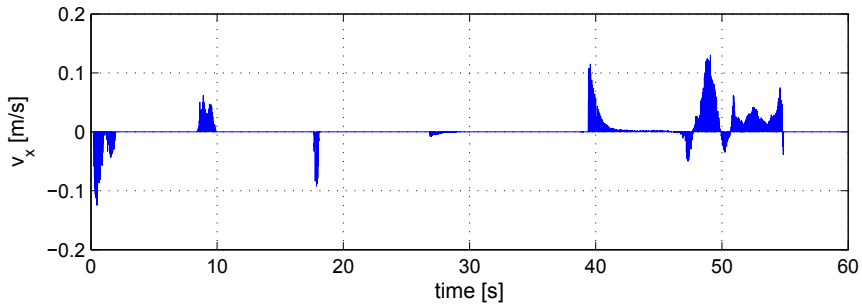


Figure 6.14: The force sensor data, $F = (F_x, F_y, F_z)$, in both real-time and playback mode.

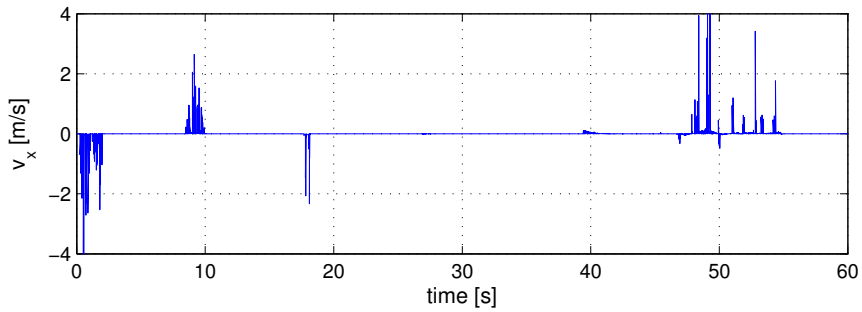
Furthermore, Figure 6.15 shows the velocity of the lower robot's tool, when running the system with the Kinect publishing data at 30 Hz. Here, (a) shows the velocity in real-time mode, while (b) and (c) shows the velocity in playback mode. The difference between Figure 6.15(b) and 6.15(c) is simply the scaling of the y-axis. There are two major observations that can be made by looking at these plots. Firstly, all the graphs are really dense. Ideally, the graphs should have had a single thin line in the x direction, instead of the shaded areas. However, because the v_x values constantly varies between zero and another value, they appear dense. The reason for these faulty zero values is that the tool center frame measurements, that are used to calculate the velocity, sometimes consist of two subsequent equal values.

In addition, the large magnitude of the graphs in Figure 6.15(b) and 6.15(c) also shows that the velocity is more noisy in playback mode. The reason for this behavior is the imprecise timing of the playback. That is, the large *.bag* files, caused by the large amount of data from the Kinect, makes the step size of the playback inconsistent. Even though Figure 6.15 is created from running the system with the Kinect at 30 Hz, a similar behavior was also seen with the Kinect at 10 Hz.

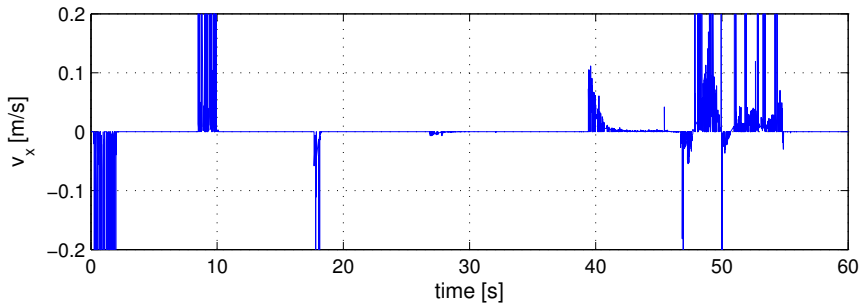
In Figure 6.16, the step size of the received data is shown. The step size, Δt, is the time between two subsequent tool center frame values that are received in the Robot Data Processor node. Here, the Kinect is used, and Figure 6.16(a) shows that the step size is consistent in real-time mode – when the data is published by the robot drivers. In Figure 6.16(b) and 6.16(c), however, the step size varies a great deal. The step size values are measured to be in the range between 0.0002 ms and 0.2 ms. Moreover, in Figure 6.16(b) and 6.16(c), the data is published by Rosbag, with Kinect data published at 30 Hz and 10 Hz, respectively. The reason for the inconsistent step sizes is the large amount of data that is stored in the *.bag* files and played back by Rosbag. As both the raw RGB and depth image from the Kinect are recorded, there is too much data for Rosbag to handle – even when the Kinect only publishes at 10 Hz.



(a) Real-time mode.

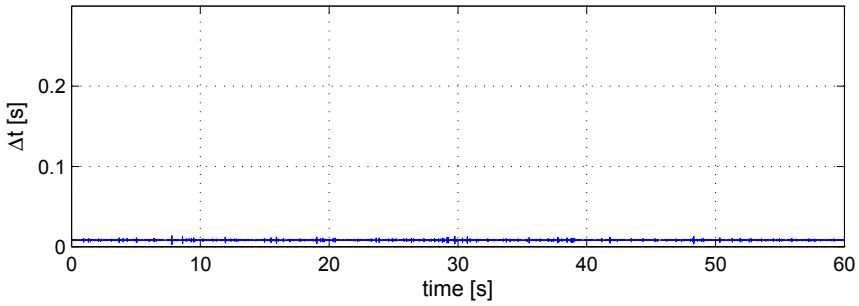


(b) Playback mode. Note the scaling on the y-axis.

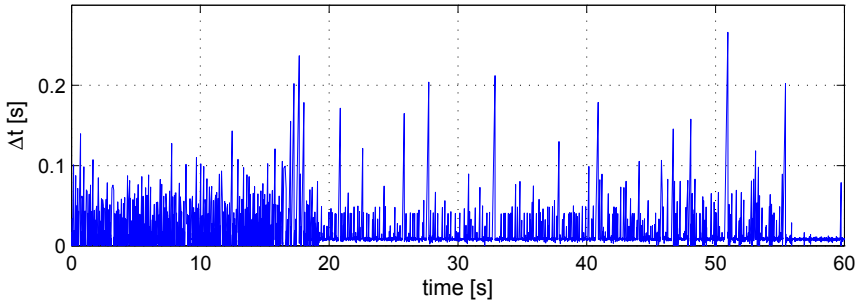


(c) Playback mode. This graph is the same as (b), but it has the same scaling on the y-axis as (a).

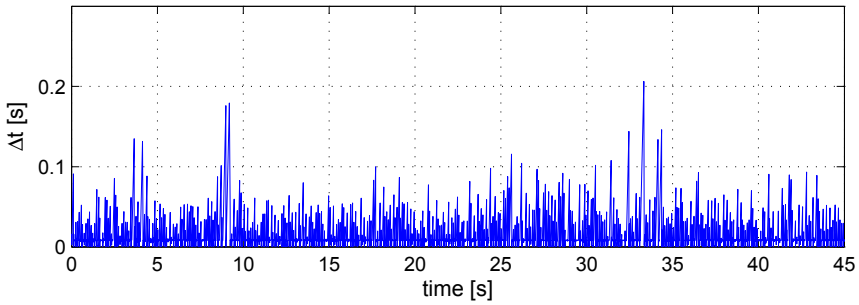
Figure 6.15: The velocity, v_x, as function of time. A similar behavior was also seen for the y and z components of v. These plots are generated by running the system with the Kinect at 30 Hz. A similar behavior was also seen with the Kinect at 10 Hz.



(a) Real-time mode, with Kinect at 30 Hz.



(b) Playback mode, with Kinect at 30 Hz.



(c) Playback mode, with Kinect at 10 Hz.

Figure 6.16: The step size, Δt, as function of time.

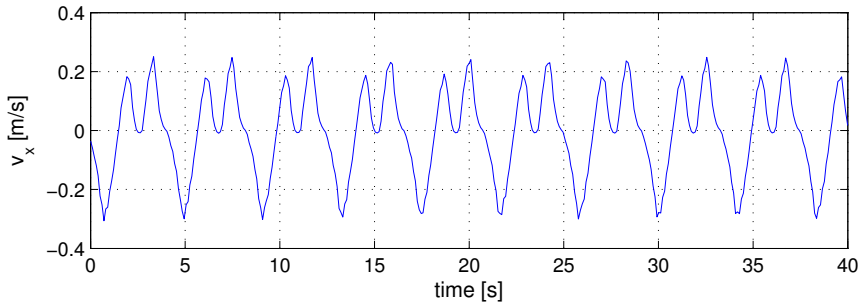
Figure 6.17 shows the test case with no Kinect. Here, Figure 6.17(a) shows the velocity of the robot tool, Figure 6.17(b) shows the step size for the tool center frame topic and Figure 6.17(c) shows the step size for the force sensor data. In these plots it can be seen that the velocity is smooth and has no sudden zero values, unlike the corresponding plot of the test cases when using the Kinect. This is because the step size is consistent, resulting in a more stable velocity. As the tool center frame data for this test case is only published at 10 Hz, Figure 6.17(c) is included to show that the system is also able to play back data at a higher frequency.

Topic / Data[1]	**Publish Rate** [Hz]	**Record Rate** [Hz]	**Playback Rate** [Hz]
Tool center frame (30 Hz)	123.5	102.8	89.0
Force sensor (30 Hz)	93.8	77.2	70.3
Tool center frame (10 Hz)	123.1	123.0	100.0
Force sensor (10 Hz)	95.7	95.2	84.8
Tool center frame (0 Hz)	9.9	9.9	9.9
Force sensor (0 Hz)	93.1	93.1	93.1

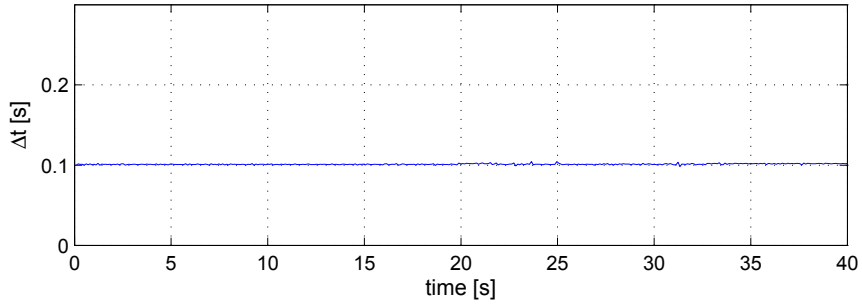
Table 6.2: Publish, record and playback rates for the tool center frame and force sensor data (topics), with the Kinect publishing at 30 Hz, 10 Hz and 0 Hz.

Table 6.2 lists the different measured rates, for the different test cases. Here, *publish rate* refers to the data published from the robot drivers, *record rate* refers to what is recorded in the *.bag* file, and *playback rate* refers to what is being played back from the *.bag* file. From this table, it can be seen that when Kinect data at 30 Hz is included, a lot of data is lost. The reason for this loss is that Rosbag is not able to process the large amount of data. Furthermore, it can also be seen that at 10 Hz Rosbag is able to record the data successfully. However, the total amount of data is still too large, resulting in an unsuccessful playback. In this case, the playback is better than at 30 Hz, but not perfect. Finally, Table 6.2 also

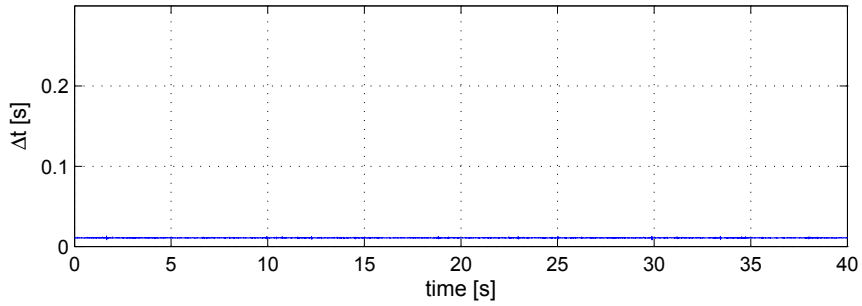
[1]The paranthesises denote the publish frequency of Kinect data; 0 Hz means that the Kinect is not used in the visualization.



(a) Velocity of the robot tool.



(b) Step size for the tool center frame data.



(c) Step size for the force sensor data.

Figure 6.17: The velocity and step size – in playback mode. Here, there is no Kinect data stored in the *.bag* file.

lists the rates when the Kinect is not used (written as 0 Hz). In this last test case, it can be seen that both the recording and playback is successful. The reason for the low rate of the tool center frame data is simply because this test was done at a different time than the other tests, with a different publish rate set for the drivers.

Chapter 7

Discussion

In this chapter, the complete system that is developed as part of this thesis is discussed. Firstly, in Section 7.1 a discussion of the implementation, testing and results for the specific case – the sewing cell – is presented. Then, Section 7.2 discusses the performance of general system, based on the general implementation and the testing done on the sewing cell.

7.1 Sewing Cell

As described in the previous chapter, the general system has been customized to work for a specific application – the sewing cell, which is a robotic process consisting of a sewing machine and two industrial robots. The robots are used to feed the sewing machine with leather covers, which are to be sewed together. Using visualization software makes it possible to debug and do detailed analysis while developing this robotic process. The aforementioned customization includes having a static model of the sewing machine, a model of the two industrial robots, Universal Robots UR-6-85-5-A, and a point cloud generated by a Kinect, in the virtual scene. Furthermore, to provide a more informative visualization, several sensors are displayed as arrow markers, as well as a corresponding textual output. These arrow markers includes the robot tool velocity and force sensor, as well as

the sewing machine feeding rate and edge sensors. The resulting visualization is covering the given specification, listed in Section 6.1, and can be seen in Figure 6.9 - 6.13.

The two industrial robots were created using URDF and Xacro, and the resulting model can be seen in Figure 6.4. In this figure, it can be seen that the models are missing the robot tool. For the real robots, there is a tool attached to the end-effector, which is used to grasp the workpieces. However, due to the current tool being a prototype, creating a model of this was not prioritized. Furthermore, to make the virtual robots appear similar to the real ones, their pose is updated by calculating the transformations between the different robot links, based on the measurements of the real robots' joint states. This is shown in Figure 6.9.

Moreover, each of the robots has a corresponding velocity and force marker, as shown in Figure 6.10. The arrow indicating the force sensor that is attached to the robot tool, is showed as a red arrow, and is generated based on the force sensor measurements. These measurements are published by the robot drivers when running the program in real-time mode, and by Rosbag in playback mode. From the experiments presented in Chapter 6, it was seen that the result of playing back the force sensor data is very similar to the originally published data in real-time mode. Figure 6.14 shows the force sensor data received by the Robot Data Processor node, in both real-time and playback mode. This data is used to calculate the direction and magnitude of the force arrow marker. As can be seen in this figure, the force data is almost identical in both program modes, which results in an equal behavior for the visualization of the force markers.

Similar to the force, the velocity is also indicated as an arrow, which is calculated based on data published by the robot drivers or Rosbag, depending on the program mode. The velocity is visualized as a green arrow, and it is calculated according to Equation (5.1). That is, based on the position measurements of the tool center frame that are received from the robot drivers (or Rosbag in playback mode), as well as the time between two subsequent measurements. Based on the experiments presented in the previous chapter, it was seen that as long as the robot drivers publish correct data, and the total amount of data to be recorded and played back

is not too high, the velocity is calculated correctly. Here, *correct data* refers to there being no false subsequent equals values, which causes the velocity to be zero. This problem is illustrated in Figure 6.15(a). Because the position measurements that are used to calculate the velocity contain some subsequent equal values, the resulting velocity has some faulty zero values. A result of this is that the graphs appear dense. However, as the velocity is only used for specifying the length and direction of the arrow marker, this problem is not really noticed when viewing the visualization.

Furthermore, as mentioned above, the playback precision is high, as long as the total amount of data flow is not too large. The estimated velocity of the robot tool is smooth and played back correctly, with no Kinect in the system, as can be seen in Figure 6.17. In this figure, a smooth velocity and consistent step size can be seen. However, with a larger data flow, due to the large amount of data published by the Kinect, some data is lost during recording and playback. In addition, the timing precision of the played back data is lower. This data loss and low precision is caused by the fact that Rosbag, which is doing the actual recording and playback, is not able to process data fast enough, resulting in an unstable recording / playback. Thus, the Robot Data Processor node does not receive data at a constant rate, which affects the estimation of the velocity. Since the messages with the sensor data that are published by the robot drivers do not contain a timestamp, the time used to calculate the step size is set in the callback function for the corresponding topics. Thus, if the data is received at a variable rate, the step size will vary. Figure 6.16 illustrates this problem, and shows the step size of the played back data when using a Kinect. The result of this is an estimate of the velocity that contains several spikes, as seen in Figure 6.15(b) and 6.15(c). These spikes are caused by a faulty large velocity, due to a faulty – too large – step size.

As long as the robot drivers publish data with no false subsequent equal values, and Rosbag is able to publish data at a constant rate, the velocity marker (arrow) is displayed correctly in both program modes. In addition to the aforementioned figures, this is also showed in Table 6.2. Here, the rate of the data published by the drivers, recorded by Rosbag, and played back by Rosbag is listed. When not using

a Kinect, all data is recorded and played back correctly. Furthermore, when using the Kinect at 10 Hz, data is recorded successfully, but not played back equally. Also, at 30 Hz, data is lost both when recording and playing back. Although the inconsistent playback applies to all the played back data, such as force sensor data and joint states, only the estimation of the velocity is affected as described above. The reason for this is that the time between two subsequent samples is used directly to calculate the velocity.

Based on experiences from the testing that was done, an idea for a new feature is to filter the calculated velocity. Currently, the velocity is calculated by looking at the change in position and time for two subsequent samples. However, the resulting velocity could be further improved by applying a smoothing filter, such as a moving average filter. Also, by making sure that the data published by the drivers are actually correct, and in addition being certain that there is not too much data for Rosbag to handle, these problems will never occur. Moreover, the problem caused by Rosbag not being able to handle all the data, can also be solved by upgrading the hardware, by using a computer that can read and write data at a higher rate.

According to the specifications, a model of the sewing machine has been included in the visualization, as well as arrow markers indicating the edge sensors and feeding speed of the sewing machine. The resulting display can be seen in Figure 6.13. Here, a purple arrow is used to indicate the feeding speed, and the cyan and yellow arrows are representing the upper and lower edge sensor, respectively. The arrows representing the edge sensors are generated based on the difference between the actual measurement and a given setpoint. For simplicity, this setpoint can be changed, by setting a parameter as described in the previous chapter.

Moreover, the implementation also includes a Kinect sensor, used to provide an image of the workpieces, or leather covers. The Kinect drivers combine RGB and depth data to generate a point cloud, which is added to the rest of the virtual scene. The result of this is an informative view of the parts that cannot be modeled. Being able to see the workpieces is important when using the visualization to study details of the sewing process. To improve the resulting visualization, the point

cloud was segmented into only consisting of the region of interest (ROI), which was defined to be a box around the area of the workpieces. Furthermore, based on the testing that was done, an idea for a future version of this program is to add additional sensors, to improve the 3D view. The testing described in this thesis used one Kinect sensor, mounted to the ceiling. This way, an informative view of the workpieces was seen by setting the camera view of the virtual scene above the process. However, due to this being a point cloud generated from one single sensor, information from different viewing angles is missing. An improvement to this could be to add additional depth sensors, placed at different angles. Thus, by calibrating the sensors, a more extended and detailed point cloud can be visualized.

One of the challenges of customizing the general process visualizer for the sewing cell was to calibrate the system. This calibration includes aligning the devices correctly, to make the virtual scene appear similar to the real one. The devices are calibrated by setting the base frame under the `<base>` tag in the XML configuration. This frame is then set relative to the global frame, which is defined to be the upper robot's base frame. Since the frames are specified in meters and radians, a suggested approach for calibrating the devices is to do measurements on the real devices, and use these values directly. Then, as long as the models are correct, the complete virtual scene will appear similar to the real one.

7.2 General System

The general system that is developed as part of this thesis, is a program that can be used for 3D visualization of robot cells. The program is able to visualize 3D models of robots and other devices, by receiving measurements of the corresponding joint states. Furthermore, the program also supports point clouds generated by depth sensors, to visualize the parts of the process that is not modeled, as well as different process variables visualized as arrows, with a corresponding textual description. For each of the virtual robots, the general system provides a visualization of the tool velocity and force sensor – if available. Moreover, the system is created in a way that makes it easy to add more information to the visualization, through the

use of a configuration file, written using a simple XML syntax. Additionally, the system has two program modes: real-time and playback mode. In *real-time* mode, the robotic process can be viewed in real-time. It is also possible to record data in this mode. The recorded data can then be played back in *playback* mode. By running a visualization in playback mode, some of the additional features, such as setting the playback speed and pausing, resuming and stepping through the data, can be used.

Due to the two program modes and all the information gathered in the visualized scene, the system presented in this thesis is a powerful tool for analyzing a robotic process. When running the program in playback mode, a closer look at the process can be made by utilizing the functionality given by the media buttons. It is possible to pause and resume the playback, as well as stepping forward. When doing the testing described in the previous chapter, this functionality was very useful. Furthermore, the practical work in the lab resulted in some new ideas for additional features in future versions. First of all, it would be useful to be able to control the speed of the playback dynamically. In the current implementation, the playback has to be stopped in order to set a different playback speed. Having a way to control the speed during runtime would enhance the usability. This also includes being able to fast-forward and rewind the playback. For instance, if something interesting happens and the human operator presses pause too late, he or she currently has to wait until the playback has looped though everything, in order to pause at the desired point in time. Adding the functionality to dynamically change the playback speed was considered. Currently, the Rosbag command-line tool is used for playing back data. This tool does not support rewinding or fast-forwarding. Thus, adding this functionality requires a complete rewrite of the current way to play back data. For instance, one approach could be to use the Rosbag code API but doing this would require a great deal of work, because of the need to do all processing of the *.bag* files directly in the code.

Another alternative approach for playing back data is by using *Rxbag* [49]. Rxbag is another standard ROS tool for playing back *.bag* files, which was considered in the beginning of this study. It consists of a nice GUI (see Figure 7.1), with different media buttons, a list of all available topics, and several options for deciding which

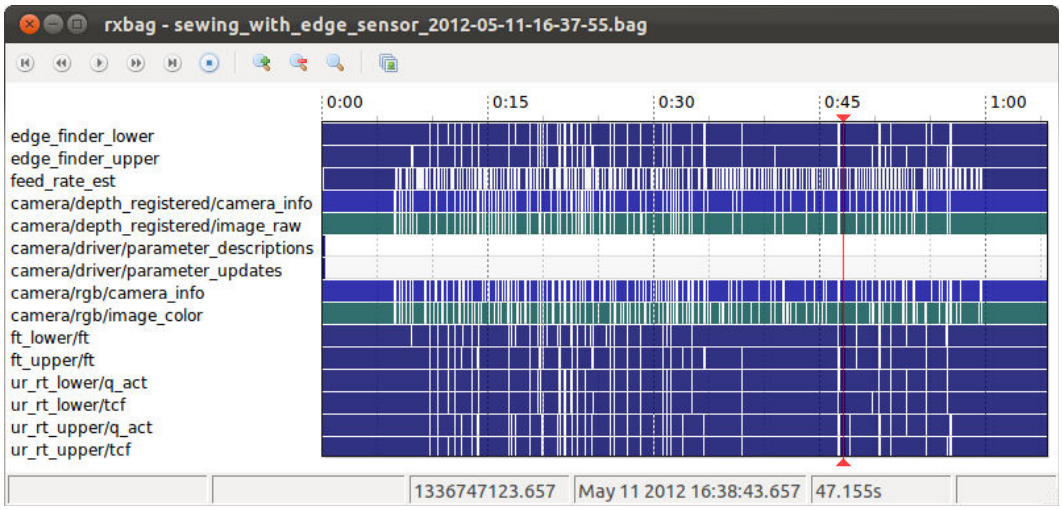


Figure 7.1: Rxbag for the *.bag* file used in some of the experiments presented in Chapter 6.

topics to publish. However, the timing precision of this tool is very poor, and it was therefore chosen not to use as the main way to playback data. In addition, it uses a lot of resources, with makes the whole system really slow. Despite this, it is still possible to use Rxbag to playback data, by launching the Process Visualizer in playback mode, without actually playing back anything, and starting Rxbag externally.

The main focus when developing the system described in this thesis was to make the program as dynamic as possible. That is, making the system as general as possible, to easily customize it for different robotic applications. To see how easily new devices can be added to an existing system, Figure 7.2 is included. Here, two additional robots are added to the rest of the sewing cell. This was done by adding two extra devices in the XML configuration, and specifying that the use the same topics to receive data as for the other robots. This way, they move similarly to the existing robots. If other real robots were available, the virtual robots could be set to use the sensor data from the real ones instead, to create an expanded visualization.

As can be seen in Figure 7.2, it is easy to specify which devices to use in a system. This is mainly because of the use of the XML configuration, which was developed

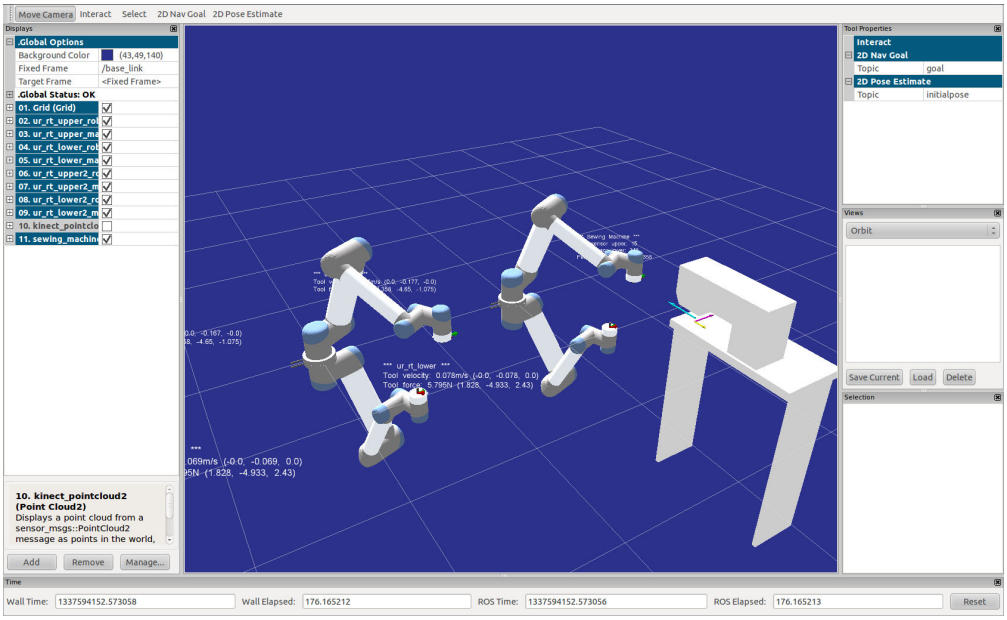


Figure 7.2: An example showing how easy it is to expand the number of devices.

as part of this thesis. The XML tags included in this implementation are added to provide the functionality necessary to meet the given specifications. For instance, the `<node>` tag makes it possible to write code to be executed for a specific device, which makes the system very flexible. Even if writing code in this way gives the operator a great deal of possibilities, it might sometimes be desirable to implement a more simple syntax at the expenses of generality. Thus, because of the easily extendable program structure, additional XML tags, such as the `<param>` tag, similar to the one used for *.launch* files, can be implemented in the future. Following the syntax of *.launch* files, this tag could be used to set parameters directly from the XML configuration, instead of having to do this in a Python script or similar. Making the syntax as similar to Roslaunch as possible is clearly beneficial, as this is known to most ROS users. Furthermore, another suggested improvement also includes being able to use substitution arguments, such as `$(find package_name)`, for which the purpose is to serve as a replacement for explicitly writing the actual path to a given package.

Another thing that was noticed when running experiments on the system, is that the markers and displays in RViz are not always updated at the same time, as

shown in Figure 6.11. The displays in RViz are drawn relative to a specific frame. Thus, because the new display information is not synchronized with the new frame transformations, the displays might be drawn in frames with different timestamp. For instance, the pose of robot models are updated as soon as new frame transformations are published by the *Robot State Publisher* node. Furthermore, the arrow markers that are used to indicate the velocity and force sensor measurement also use the robot frames, in order to be drawn at the position of the robot tool. However, these displays are not updated synchronously, Because of this, the robots are sometimes updated after the arrows, meaning the arrows will lag behind. As an example, if the update frequency of the velocity marker and robot frames are both 50 Hz, and the robot frames are updated 1 ms after the velocity marker, the velocity marker will be visualized in the wrong position for $\frac{1000}{50}$ ms -1 ms $= 19$ ms. A suggested approach for improving this it to synchronize all the published sensor data. This can be done by calculating all frame transformations in the *Robot Data Processor* node, instead of using *Robot State Publisher*, and publishing all transformations and marker data at the same time. In the program described in this thesis, the synchronization problem is minimized, by publishing all markers as soon as one single new measurement is available. Then, when RViz receives the new data, it will use the most recent frame to update the display. Since both data from the force sensor and tool center frame (used for calculating the tool velocity) are received at a rate of about 100 Hz, new makers are published every $\frac{1}{100 \cdot 2}$ ms $= 5$ ms for each robot. For a system consisting of several robots (and other devices), this could easily cause too much necessary data processing for the CPU to handle.

When running the testing described earlier, two different versions of ROS were used. For the testing done in the laboratory, ROS Fuerte[1] was used, and for the rest, ROS Electric[2]. In both of these versions, the program behaved mostly the same, except for two issues. First of all RViz was very unstable in Electric, causing the program to crash very often, due to a problem with the graphics. However, in Fuerte this problem was not noticed at all. This is most likely due to the fact that

[1]ROS Fuerte: At time of writing, this is the latest version of ROS [50]; it was released on April 23, 2012.

[2]ROS Electric: A version of ROS, released on August 30, 2011 [51].

RViz is rewritten in Fuerte. The old version of RViz used wxWidgets and OGRE, while the new version uses the QT framework [50]. Additionally, another different behavior in the two versions is the Kinect drivers. In Electric these drivers worked perfectly, but in Fuerte there is a problem where one of the processes does not exit when it should have. All the Kinect nodes are launched by `openni_launch`, which is started as a subprocess of the Launcher GUI. These nodes are then also killed by the GUI node as directed by the human operator. However, in Fuerte one of these nodes does not always exit, causing an erroneous restart of the drivers. This was temporarily fixed by manually killing the process that caused the problem. Most likely, this problem will be fixed in a future release of the drivers.

Chapter 8

Conclusion and Further Work

8.1 Conclusion

In this thesis, an approach for visualizing robot cells has been proposed. First, a literature survey of currently available software was made. Based on this survey, a way to visualize a process consisting of robots and other devices has been proposed and developed. This development uses Robot Operating System (ROS) and the visualization environment RViz as basis for the software, and it is designed to support different kinds of robots, depth sensors and other devices.

The software developed for this thesis consists of a Graphical User Interface (GUI), which is used to control the visualization of a robotic process. This GUI provides functionality for selecting which devices to visualize, as well as functionality for recording and playing back data. All the available devices in the system, and different information about these, are specified in a XML configuration file. By running the program in real-time, different measurement data for the devices can be recorded. This data can then be played back later, to visualize the same process offline. Playing back previously recorded data makes it possible to freeze the visualization, by pausing the playback, or to step through the data one value at a time. The actual visualization of the process is done in RViz, which is configured dynamically by the GUI program based on information specified by a human

operator. Here, it is possible to move the camera view, and to zoom in on specific parts of the process, for instance to view the representation of a specific process variable.

To test the design described in this thesis on a specific application, the program was customized for the Ekornes sewing cell at SINTEF Raufoss Manufacturing. This customization included the generation of a rather detailed XML configuration file with information about the devices used in the process. These devices are a sewing machine, two industrial robots and a Kinect sensor. The Kinect sensor is used to visualize the dynamic parts of the process, that is, the workpieces. In addition, several arrows, and a corresponding textual description, are used to indicate different sensor data for both robots and the sewing machine.

Based on several experiments that were run on the system, it was seen that the program can be used as a powerful analyzing tool, because of the different program modes, and the informative visualization view. As long as the total amount of data flow is not too high, the precision of the playback is high, which results in a visualization that appear equal in both program modes, and similar to the real process. However, the amount of data required to generate a 3D point cloud, is too large for the current system to handle on a standard desktop computer. Thus, a completely correct playback requires either better hardware, or to reduce the publish rate of the depth sensor used to generate the point cloud.

8.2 Further Work

Based on everything described in Chapter 7, it might be desirable to improve the current system even further. First of all, there could be a way to rewind, fast-forward and dynamically change the speed when playing back data. In the current implementation, the Rosbag command-line tool is used for playing back data. By using this tool, implementing the desired functionality is not possible. Thus, the whole playback part of the program has to be rewritten, for instance by using the Rosbag code API. Another alternative is to use a different ROS tool called Rxbag for the playback. However, even though this tool provides rewind and fast-

forward functionality, the timing precision in the playback is rather poor. Thus, using Rxbag is only recommended if this issue is fixed in a future release.

Moreover, Chapter 6 showed that for a large total amount of data flow in the system, the precision of the playback is rather low. An idea for a new feature to improve this, is to add a filter, such as a moving average filter, to smooth the estimate of the velocity. For the application described in this thesis, the issues regarding timing precision did not really cause any problems. However, other applications might require a higher precision. Using a filter will also reduce the faulty zero values behavior, caused by the subsequent equal position values, for the estimated velocity.

In addition to this, another suggested approach is to make sure that all the displays in RViz are synchronized and updated at the same time, to prevent devices from lagging behind the others. As described in Chapter 7, this can be done by calculating all the robot frame transformations in the *Robot Data Processor* node, and use this information when setting the position and frame of the velocity and force markers.

Furthermore, the system could also benefit from having more supported XML tags to use in the configuration file. An example tag that could be included is `<param>`, similarly to the tag used for ROS *.launch* files[1]. The purpose of this tag is to set the value of a variable on the parameter server[2]. In the program described in this thesis, parameters have to be set by doing so explicitly in the program code of a node.

Another proposal for further work is to make the point cloud segmentation a part of the *Depth Data Processor* node (see details about this in Chapter 5). The specific implementation for the sewing cell, described in Chapter 6, contains a node called *Kinect ROI*, which segments the point cloud from the Kinect, to get a resulting point cloud from the region of interest (ROI) only. This segmentation is done by using the Point Cloud Library (PCL), and the node doing this processing must therefore be written in C++, since PCL is not supported in Python. Thus,

[1] *.launch* files are used to launch several ROS nodes at the same time, using a standard ROS tool called *Roslaunch*

[2] Parameter server: See Section 3.1 for a description

implementing this as a part of the Depth Data Processor requires rewriting it in C++, which again requires a new XML parser in C++ for the configuration files. An alternative is to use a binding, or to call the C++ program from Python.

In the system described in this thesis, all devices have to be calibrated manually offline. That is, the devices' position and orientation relative to the global frame, have to be specified in the XML configuration file. Moreover, when making changes to the configuration file, the program has to be restarted to reload the new configuration. A further improvement to this, is to do the calibration in the GUI, by having a possibility to set the frame transformation while the program is running. Also, to be able to reload the XML configuration file without restarting the program could be useful. This functionality could be implemented simply by adding a button that updates the GUI and all corresponding objects in the program.

The implementation that was done for the sewing cell used a Kinect sensor to generate depth data of the workpieces. From this depth data, a point cloud was generated and visualized in RViz. As the camera was mounted near the ceiling, pointing down at the process, the workpieces can be seen clearly when viewing the visualization from above. However, if the camera view in RViz is changed, the workpieces can be seen less clearly, due to the lacking information in the point cloud. An approach for improving this is to use additional Kinect sensors. Then, a larger and more dense point cloud can be generated, by placing the sensors at different locations and combining the resulting data.

Bibliography

[1] A. Harris and J. Conrad. "Survey of popular robotics simulators, frameworks, and toolkits". In: *Southeastcon, 2011 Proceedings of IEEE*. 2011, pp. 243 – 249.

[2] *Robot Operating System (ROS)*. Retrieved February 9, 2012. URL: `http://ros.org/`.

[3] M. Quigley et al. "ROS: an open-source Robot Operating System". URL: `http://pub1.willowgarage.com/~konolige/cs225B/docs/quigley-icra2009-ros.pdf`.

[4] J. Kramer and M. Scheutz. "Development Environments for Autonomous Mobile Robots: A Survey". In: (2007).

[5] *RViz*. Retrieved February 23, 2012. URL: `http://www.ros.org/wiki/rviz`.

[6] F. Ferland et al. "Teleoperation of AZIMUT-3, an Omnidirectional Non-holonomic Platform with Steerable Wheels". In: *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*. 2010, pp. 2515 –2516.

[7] P. Jayasekara et al. "Simultaneous localization assistance for multiple mobile robots using particle filter based target tracking". In: *Information and Automation for Sustainability (ICIAFs), 2010 5th International Conference on*. 2010, pp. 469 –474.

[8] Microsoft. *Kinect*. Retrieved March 26, 2012. URL: `http://www.xbox.com/kinect`.

[9] D. Schmidt and F. Buschmann. "Patterns, frameworks, and middleware: their synergistic relationships". In: *Software Engineering, 2003. Proceedings. 25th International Conference on*. May 2003.

[10] *CARMEN*. Retrieved February 14, 2012. URL: http://carmen.sourceforge.net/.

[11] *Player/Stage*. Retrieved February 14, 2012. URL: http://playerstage.sourceforge.net/.

[12] *Pyro*. Retrieved February 14, 2012. URL: http://pyrorobotics.org/.

[13] S. Thierfelder et al. *Robbie: A Message-based Robot Architecture for Autonomous Mobile Systems*. Tech. rep. Institute for Computational Visualistics, University of Koblenz-Landau, 2011.

[14] J. Jackson. "Microsoft Robotics Studio: A Technical Introduction". In: *Robotics Automation Magazine, IEEE* 14.4 (2007), pp. 82 –87.

[15] *Stanford AI Laboratory*. Retrieved February 9, 2012. URL: http://ai.stanford.edu/.

[16] *ROS (Robot Operating System)*. Retrieved February 9, 2012. URL: http://en.wikipedia.org/wiki/ROS_(Robot_Operating_System).

[17] *Willow Garage*. Retrieved February 9, 2012. URL: http://www.willowgarage.com/.

[18] K. Wyrobek et al. "Towards a Personal Robotics Development Platform: Rationale and Design of an Intrinsically Safe Personal Robot". In: *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*. 2008, pp. 2165 –2170.

[19] M. Quigley, E. Berger, and A. Y. Ng. "STAIR: Hardware and Software Architecture". In: (2008).

[20] S. Arias et al. "Evolution of the robotic control frameworks at INRIA Rhône-Alpes". In: *6th National Conference on Control Architectures of Robots*. INRIA Grenoble Rhône-Alpes. May 2011, 8 p.

[21] M. Montemerlo, N. Roy, and S. Thrun. "Perspectives on Standardization in Mobile Robot Programming: the Carnegie Mellon Navigation (CARMEN) Toolkit". In: *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*. Vol. 3. 2003, 2436 –2441 vol.3.

[22] B. P. Gerkey, R. T. Vaughan, and A. Howard. "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems". In: *In Proceedings*

of the 11th International Conference on Advanced Robotics. 2003, pp. 317–323.

[23] *Gazebo*. Retrieved February 16, 2012. URL: http://gazebosim.org/.

[24] D. Blank et al. "Pyro: A Python-based Versatile Programming Environment for Teaching Robotics". In: *J. Educ. Resour. Comput.* 4 (3 2003). ISSN: 1531-4278.

[25] *RoboCup@Home*. Retrieved February 13, 2012. URL: http://www.robocupathome.org/.

[26] *RoboCupRescue*. Retrieved February 13, 2012. URL: http://www.robocuprescue.org/.

[27] *SICK Robot Day*. Retrieved February 13, 2012. URL: http://www.sick.com/group/EN/home/pr/events/robot_day/Pages/robot_day_overview.aspx.

[28] D. Huber et al. "Real-time Photo-realistic Visualization of 3D Environments for Enhanced Tele-operation of Vehicles". In: *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*. 2009, pp. 1518 –1525.

[29] *transforms (tf)*. Retrieved February 23, 2012. URL: http://www.ros.org/wiki/tf/.

[30] D. Gossow et al. "Interactive Markers: 3-D User Interfaces for ROS Applications". In: *Robotics Automation Magazine, IEEE* 18.4 (2011), pp. 14 – 15.

[31] R. Bilit. "Using Sensors for Real-Time Robot Control". Project Report.

[32] *Rosbag*. Retrieved April 23, 2012. URL: http://www.ros.org/wiki/rosbag.

[33] *Point Cloud Library*. Retrieved April 17, 2012. URL: http://pointclouds.org/.

[34] *OpenCV*. Retrieved April 17, 2012. URL: http://opencv.willowgarage.com/wiki/.

[35] *URDF*. Retrieved March 21, 2012. URL: http://www.ros.org/wiki/urdf.

[36] *Xacro*. Retrieved March 22, 2012. URL: http://www.ros.org/wiki/xacro.

[37] M. Spong, S. Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. John Wiley & Sons, 2006. ISBN: 9780471649908.

[38] O. Egeland and J. Gravdahl. *Modeling and Simulation for Automatic Control*. Marine Cybernetics, 2002. ISBN: 9788292356012.

[39] *Denavit-Hartenberg parameters*. Retrieved March 21, 2012. URL: http://en.wikipedia.org/wiki/Denavit-Hartenberg_parameters.

[40] *Kinect*. Retrieved March 26, 2012. URL: http://en.wikipedia.org/wiki/Kinect.

[41] Ø. Skotheim. "Kinect Sensor - Preliminary study". Memo.

[42] *OpenKinect*. Retrieved March 26, 2012. URL: http://openkinect.org/wiki/Main_Page.

[43] *OpenNI*. Retrieved March 26, 2012. URL: http://www.openni.org/.

[44] *PrimeSense*. Retrieved March 26, 2012. URL: http://www.primesense.com/.

[45] *PySide*. Retrieved April 22, 2012. URL: http://www.pyside.org/.

[46] Euclideanspace. *Axis-angle to Quaternion Conversion*. Retrieved May 18, 2012. URL: http://www.euclideanspace.com/maths/geometry/rotations/conversions/angleToQuaternion/index.htm.

[47] J. Schrimpf and L. E. Wetterwald. "Experiments towards Automated Sewing with a Multi-Robot System". In: *International Conference on Robotics and Automation (ICRA 2012) (accepted)*. May 2012.

[48] *Universal Robot*. Retrieved May 27, 2012. URL: http://www.universal-robots.com/.

[49] *Rxbag*. Retrieved May 22, 2012. URL: http://www.ros.org/wiki/rxbag.

[50] *ROS Fuerte*. Retrieved May 22, 2012. URL: http://ros.org/wiki/fuerte.

[51] *ROS Electric*. Retrieved May 22, 2012. URL: http://ros.org/wiki/electric.

[52] *Dynamic Reconfigure*. Retrieved June 2, 2012. URL: http://ros.org/wiki/dynamic_reconfigure.

Appendix A

CD

The attached CD contains the `process_visualizer` and `sewing_cell` packages. Below is a list of the packages, and the interesting directories, files and directories in each of them. The source code can be found in the `/src/` directory, and is shortly described below.

`process_visualizer`:

- `devices.xml`: Default XML configuration, used for the sewing cell.
- `/launch/`
  - `gui.launch`: File used to launch the GUI correctly.
- `/rviz_config/`
  - `default.vcg`: File containing the default RViz configurations, which is needed to successfully create the resulting configuration.
- `/src/`
  - `bagplayer.py`: Contains functionality for playing back data from *.bag* files, using the `BagPlayer` class.
  - `bagrecorder.py`: Contains functionality for recording data to *.bag* files, using the `BagRecorder` class.

  - `depth_proc.py`: Contains of the `DepthProcessor` class, used to process data from depth sensors.
  - `gui.py`: The main Launcher GUI, implemented as the `LauncherApp` class.
  - `robot_proc.py`: Contains of the `RobotProcessor` class, used to process data from robots.
  - `rvizconfig.py`: Node used to dynamically create RViz configuration files, through the use of the `RVizConfig` class.
  - `xmlparser.py`: Module used to parse XML configuration files that are created according to the specifications in Section 5.2.1. Contains the `XmlParser` class.

`sewing_cell`:

- `/launch/`
  - `openni_live.launch`: File used to launch OpenNI/Kinect drivers in real-time mode.
  - `openni_play.launch`: File used to launch OpenNI/Kinect drivers in playback mode.
- `/meshes/`: Directory containing all visual elements of the sewing machine and robot models.
- `/src/`
  - `kinect_roi.cpp`: C++ file for the node used to segment the region of interest for the Kinect point cloud.
  - `sewing_machine.py`: The Sewing Machine Data Processor node, including the `SewingTransformer` class.
- `/urdf/`: Directory containing the URDF / Xacro models.

Appendix B

Abbreviations

Abbreviation	Description
2D	Two-dimensional
3D	Three-dimensional
API	Application Programming Interface
CARMEN	CARnegie Mellon Robot Navigation
CPU	Central Processing Unit
DH	Denavit-Hartenberg (convention)
DoF	Degrees of Freedom
ICR	Instantaneous Center of Rotation
IP	Internet Protocol
IPC	Inter-Process Communication
IR	Infrared
MSRS	Microsoft Robotics Studio

Abbreviation	Description
OpenCV	Open source Computer Vision
OpenNI	Open source Natural Interaction
PCL	Point Cloud Library
PTZ	Pan Tilt Zoom (camera)
RGB	Red Green Blue (color model)
ROI	Region of Interest
ROS	Robot Operating System
SOAP	Simple Object Access Protocol
STAIR	STanford Artificial Intelligence Robot
TCP	Transmission Control Protocol
URDF	Unified Robot Description Format
VPL	Visual Programming Language
XML	Extensible Markup Language

Appendix C

How to Run the Program

The program developed as part of this thesis consists of two ROS packages, called `process_visualizer` and `sewing_cell`. The first of these two contains the general system, and the latter contains the specific nodes to be used for the sewing cell. This chapter gives a short and informative introduction to how the program can be run.

C.1 Dependencies

Other ROS stacks and packages that are necessary to run the `process_visualizer` successfully, are as follows:

- `visualization` (stack)
- `ros_comm` (stack)
- `common_msgs` (stack)
- `tf` (package)
- `robot_state_publisher` (package)

Similarly, the `sewing_cell` depends on the following packages:

- `process_visualizer` (package)
- `pcl`
- `pcl_ros`

C.2 Launching the Program

The program can be started with the following command:

```
roslaunch process_visualizer gui.py config:=<path>
```

where `<path>` is the path to the XML configuration file. If no `config` argument is specified, the default configuration will be selected. The default configuration is set to `process_visualizer/devices.xml`

C.3 Other Tips

By default, the RGB and depth image from the Kinect is not calibrated, which results in a point cloud where the points have a faulty RGB value. Setting the `depth_registration` parameter makes the drivers use a standard calibration. This can be done either by using the `reconfigure_gui` tool [52], or by the following command:

```
rosparam set /camera/driver/depth_registration True
```

The parameter name used here might differ, depending on which version of the drivers are used. In the `openni_launch` package on ROS Electric the standard name for the parameter is as stated above. Furthermore, setting the Kinect publish rate for all images, can be done by changing the `data_skip` parameter. For instance:

```
rosparam set /camera/driver/data_skip 2
```

This will result in skipping 2 images, for every image published, which means publishing at 10 Hz. Default value is 0, corresponding to a 30 Hz publish rate.

Appendix D

Sewing Cell Details

D.1 Xacro / URDF Model

ur_rt_lower.urdf.xacro:

```
<robot xmlns:xacro="http://www.ros.org/wiki/xacro"
       name="ur_rt_lower">
    <xacro:property name="name" value="ur_rt_lower" />
    <include filename="$(find sewing_cell)/urdf/ur.urdf.xacro" />
</robot>
```

ur.urdf.xacro:

```
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
    <xacro:property name="pi" value="3.14159265" />

    <xacro:macro name="ur_link" params="id xyz rpy">
        <link name="/${name}/link${id}">
            <visual>
                <origin xyz="${xyz}" rpy="${rpy}"/>
                <geometry>
                    <mesh filename="package://sewing_cell/
                                    meshes/ur-link${id}.dae"/>
                </geometry>
```

```
        </visual>
        <collision>
            <geometry>
                <mesh filename="package://sewing_cell/
                                meshes/ur-link${id}.dae"/>
            </geometry>
        </collision>
    </link>
</xacro:macro>

<xacro:macro name="ur_joint" params="from to xyz rpy type">
    <joint name="/${name}/joint_${from}_${to}" type="${type}">
        <parent link="/${name}/link${from}"/>
        <axis xyz="0 0 1" />
        <child link="/${name}/link${to}"/>
        <origin xyz="${xyz}" rpy="${rpy}" />
    </joint>
</xacro:macro>

<!--
REMEMBER: Link 4 is fixed to Link 3 !!
Thus, numbering does not directly correspond to the
DH parameters in the table in Chapter 6.
-->

<xacro:ur_link id="0" xyz="0 0 0"
               rpy="0 0 0" />
<xacro:ur_link id="1" xyz="0 0 0"
               rpy="0 0 0" />
<xacro:ur_link id="2" xyz="0.0684 0 0"
               rpy="${-pi/2} 0 ${pi/2}" />
<xacro:ur_link id="3" xyz="0 0.425 0.061"
               rpy="0 0 ${pi/2}" />
<xacro:ur_link id="4" xyz="0 0 0.061"
               rpy="0 0 0" />
<xacro:ur_link id="5" xyz="0 0 0.0638"
               rpy="0 0 ${+pi/2}" />
<xacro:ur_link id="6" xyz="-0.0453 0 0"
               rpy="${pi/2} ${pi} ${+pi/2}" />
<xacro:ur_link id="7" xyz="0 0.0453 0"
```

```
                    rpy="${pi/2} 0 0" />

    <xacro:ur_joint from="0" to="1" xyz="0 0 0"
                    rpy="0 0 0" type="continuous" />
    <xacro:ur_joint from="1" to="2" xyz="0 0 0.0892"
                    rpy="${pi/2} 0 0" type="continuous" />
    <xacro:ur_joint from="2" to="3" xyz="-0.425 0 0"
                    rpy="0 0 0" type="continuous" />
    <xacro:ur_joint from="3" to="4" xyz="0 0 0"
                    rpy="0 0 0" type="fixed" />
    <xacro:ur_joint from="4" to="5" xyz="-0.39243 0 0"
                    rpy="0 0 0" type="continuous" />
    <xacro:ur_joint from="5" to="6" xyz="0 0 0.109"
                    rpy="${pi/2} 0 0" type="continuous" />
    <xacro:ur_joint from="6" to="7" xyz="0 0 0.093"
                    rpy="${-pi/2} 0 0" type="continuous" />

    <link name="/${name}/tool">
        <visual>
            <geometry>
                <cylinder radius=".005" length="0.05"/>
            </geometry>
            <material name="dark">
                <color rgba="0.4 0.4 0.4 1.0"/>
            </material>
        </visual>
        <collision>
            <geometry>
                <cylinder radius=".01" length="0.005"/>
            </geometry>
        </collision>
    </link>

    <joint name="/${name}/joint_7_tool" type="continuous">
        <parent link="/${name}/link7"/>
        <axis xyz="0 0 1" />
        <child link="/${name}/tool"/>
        <origin xyz="0 0 0.082" rpy="0 0 0" />
    </joint>
</robot>
```

D.2 List of Nodes

This section lists all the interesting nodes that are used in the system customized for the sewing cell, as well as which topics and parameters they use. For simplicity, *Rosbag* and all the static transformer and standard Kinect nodes are excluded. In real-time mode, all the topic messages are published by the corresponding drivers, while in playback mode they are published by Rosbag. This list of nodes is included here to give a better understanding of the general system and the sewing cell customization.

The nodes developed during this study are:

`process_visualizer`: The Process Visualizer Launcher GUI

Subscribed topics:

`clock` (rosgraph_msgs/Clock)

The current playback time; published by Rosbag

`ur_rt_upper/robot_proc`: Robot Data Processor for the upper robot

Subscribed topics:

`ur_rt_upper/q_act` (std_msgs/Float32MultiArray)
`ur_rt_upper/tcf` (std_msgs/Float32MultiArray)
`ft_upper/ft` (std_msgs/Float32MultiArray)

Published topics:

`ur_rt_upper/joint_states` (sensor_msgs/JointState)
`ur_rt_upper/marker_array` (visualization_msgs/MarkerArray)

Parameters:

`process_visualizer/playback_rate` (double)

`ur_rt_lower/robot_proc`: Robot Data Processor for the lower robot

Subscribed topics:

`ur_rt_lower/q_act` (std_msgs/Float32MultiArray)
`ur_rt_lower/tcf` (std_msgs/Float32MultiArray)
`ft_lower/ft` (std_msgs/Float32MultiArray)

Published topics:

`ur_rt_lower/joint_states` (sensor_msgs/JointState)
`ur_rt_lower/marker_array` (visualization_msgs/MarkerArray)

Parameters:

`process_visualizer/playback_rate` (double)

`kinect/depth_proc`: Depth Data Processor for the Kinect sensor

Subscribed topics:

`kinect/roi_points` (sensor_msgs/PointCloud2)

Published topics:

`kinect/points` (sensor_msgs/PointCloud2)

`kinect/kinect_roi`: Kinect ROI node, used to segment the region of interest

Subscribed topics:

`camera/depth_registered/points` (sensor_msgs/PointCloud2)

Published topics:

`kinect/roi_points` (sensor_msgs/PointCloud2)

`sewing_machine/sewing_machine`: Sewing Machine Data Processor

Subscribed topics:

`edge_finder_upper` (std_msgs/UInt16)
`edge_finder_lower` (std_msgs/UInt16)
`feed_rate_est` (std_msgs/Float32)

Published topics:

`sewing_machine/marker_array` (visualization_msgs/MarkerArray)

Parameters:

`sewing_machine/setpoint_upper` (double)
`sewing_machine/setpoint_lower` (double)

Other nodes that are used:

`ur_rt_upper/robot_state_publisher`: Robot State Publisher for the upper robot

Subscribed topics:

`ur_rt_upper/joint_states` (sensor_msgs/JointState)

Published topics:

`tf` (tf/tfMessage)

Parameters:

`ur_rt_upper/robot_description` (urdf map)

`ur_rt_lower/robot_state_publisher`: Robot State Publisher for the lower robot

Subscribed topics:

`ur_rt_lower/joint_states` (sensor_msgs/JointState)

Published topics:

`tf` (tf/tfMessage)

Parameters:

`ur_rt_lower/robot_description` (urdf map)

`rviz`: RViz – the actual visualization node

Subscribed topics:

`tf` (tf/tfMessage)
`ur_rt_upper/marker_array` (visualization_msgs/MarkerArray)
`ur_rt_lower/marker_array` (visualization_msgs/MarkerArray)
`kinect/points` (sensor_msgs/PointCloud2)
`sewing_machine/marker_array` (visualization_msgs/MarkerArray)

D.3 Frames

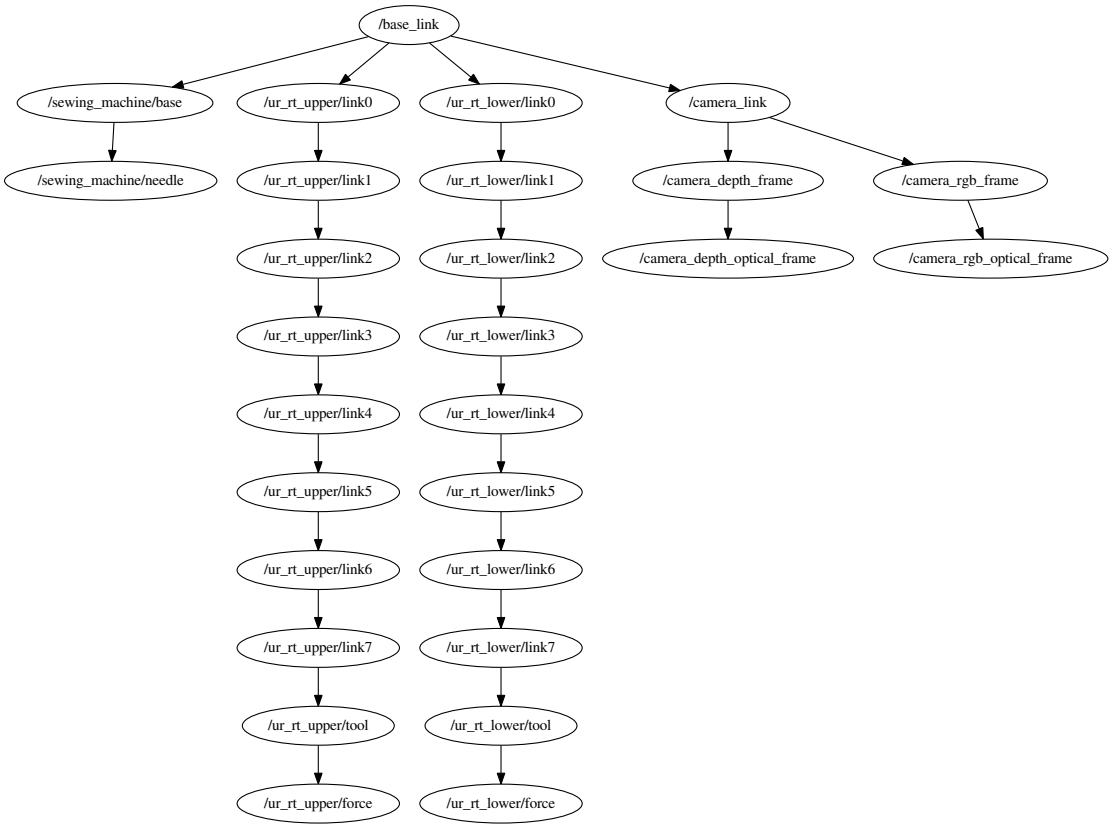


Figure D.1: An overview of all the frames for the sewing cell devices.